Chapter 5: Polynomials and Sparse Matrices

Data Structures and Algorithms with C

Polynomial ADT · Array of terms · padd · Multiplication with lists · Triples · transpose · fast_transpose · Sparse multiplication · CSR

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/dsa_c/slides/Chapter_05_Polynomials_and_Sparse_Matrices.html

Learning Outcomes

Choosing a representation

Compare a coefficient array and an array of terms for polynomials in space and time.

Merge logic

Add two sorted term arrays with padd in O(n + m) and analyze it.

Multiplying with lists

Multiply polynomials stored as linked lists, combining terms with equal exponents.

Sparse matrices

Store a matrix as ⟨row, col, value⟩ triples and say when this pays off.

Transposing

Trace transpose and fast_transpose; compare O(cols·terms) with O(cols + terms).

Multiplying

Build sparse matrix multiplication with the help of a transpose; get to know CSR.

A Polynomial: a Set of ⟨Exponent, Coefficient⟩ Pairs

$A(x) = a_n x^n + \ldots + a_1 x + a_0$; each term $a_i x^i$ has a **coefficient** and an **exponent**. Degree: the largest exponent with a nonzero coefficient.

Operation	Meaning
`Zero()`, `IsZero(p)`	build / test the zero polynomial
`Coef(p, e)`	coefficient of the term with exponent e (0 if none)
`LeadExp(p)`	largest exponent
`Attach(p, c, e)`, `Remove(p, e)`	add / remove a term
`SingleMult(p, c, e)`	$p \cdot c\, x^e$
`Add(p1, p2)`, `Mult(p1, p2)`	sum, product

Addition

$A(x) + B(x) = \Sigma\, (a_i + b_i) \cdot x^i$: coefficients of equal exponents are added.

Multiplication

$A(x) \cdot B(x) = \Sigma\, (a_i x^i \cdot \Sigma\, b_j x^j)$: exponents add, coefficients multiply.

Method 1 (Bad): Keep Every Coefficient

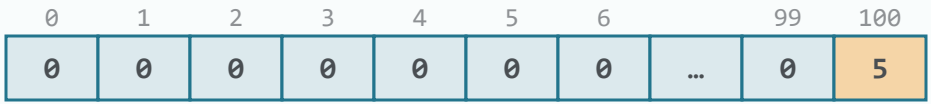
```c
#define MAX_DEGREE 101   /* max degree + 1 */
typedef struct {
  int degree;               /* largest exponent */
  float coef[MAX_DEGREE];   /* coef[i]: coef of x^i */
} polynomial;
polynomial P;
```

Pro

With `degree` stored, the degree is not recomputed each time; Coef and addition are very simple.

Con

Terms with coefficient 0 are stored too: a big **waste of memory** for a sparse polynomial.



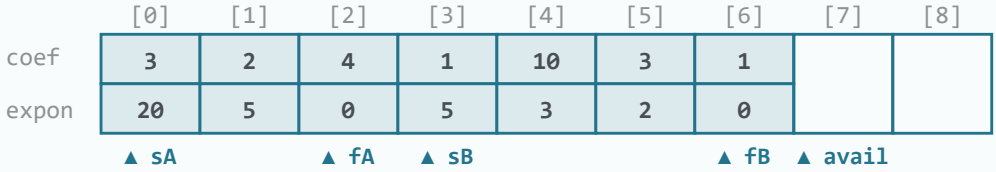
For $P(x) = 5x^{100}$, coef[0..99] hold useless zeros: 101 floats, 1 useful

Method 2 (Better): Only the Nonzero Terms

```
#define MAX_TERMS 101   /* max number of terms */
typedef struct {
  float coef;            /* coefficient */
  int expon;             /* exponent */
} polynomial;
polynomial P[MAX_TERMS]; /* all polynomials here */
int avail = 0;           /* first free slot */
```

Terms are kept in **decreasing exponent** order; each polynomial is a range `[start, finish]` of P.

> **Instructor** *"Merge logic lets us build more efficient algorithms on sorted structures."*



$$A(x) = 3x^{20} + 2x^{5} + 4 \cdot B(x) = x^{5} + 10x^{3} + 3x^{2} + 1$$

Adding by Merging with Two Pointers

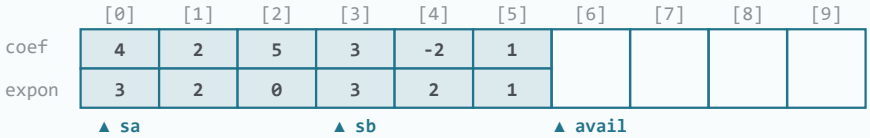
- **Equal exponents**: add the coefficients; if the sum ≠ 0, append it to the result; **both pointers** advance.
- **A's is larger**: "B has no term with A's exponent" — B's remaining terms are all smaller. Append A's term; only A advances.
- **B's is larger**: symmetric; append B's term.
- When one runs out, **copy the rest** of the other as is.

```
#define COMPARE(x, y) \
  ((x) < (y) ? -1 : (y) < (x) ? 1 : 0)

void attach(float coef, int expon) {
  if (avail >= MAX_TERMS) {
    fprintf(stderr, "Too many terms!\n"); exit(1);
  }
  P[avail].coef = coef;
  P[avail++].expon = expon;
}
#define COPY(k) attach(P[k].coef, P[k].expon)
```

Writing `attach(int coef, …)` **truncates** a float coefficient to int (2.5 → 2). The parameter must be `float`.

padd: A + B, Term by Term



A = $4x^3 + 2x^2 + 5$ → P[0..2], B = $3x^3 - 2x^2 + x$ → P[3..5]. The result D starts at **avail = 6**: *sd = 6.

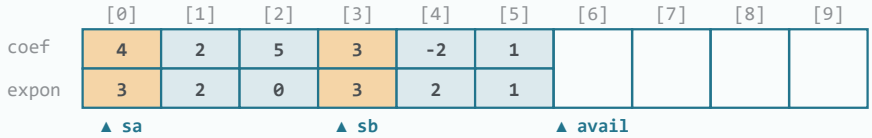
Step 1 / 14

```
1  void padd(int sa,int fa,int sb,int fb,int*sd,int*fd){
▶    float c; *sd = avail;              /* start of D */
3    while (sa <= fa && sb <= fb)
4      switch (COMPARE(P[sa].expon, P[sb].expon)) {
5      case -1: COPY(sb); sb++; break;   /* B larger */
6      case  1: COPY(sa); sa++; break;   /* A larger */
7      case  0: c = P[sa].coef + P[sb].coef;
8        if (c) attach(c, P[sa].expon); /* skip 0 */
9        sa++; sb++;
10     }
11   for (; sa <= fa; sa++) COPY(sa);    /* rest of A */
12   for (; sb <= fb; sb++) COPY(sb);    /* rest of B */
13   *fd = avail - 1;
14 }
```

VARIABLES

sa = 0 | sb = 3 | avail = 6

padd: A + B, Term by Term



COMPARE(3, 3) = **0**: equal exponents.

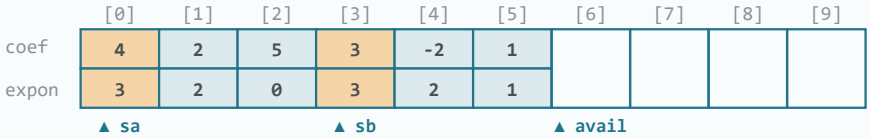
Step 2 / 14

```
1  void padd(int sa,int fa,int sb,int fb,int*sd,int*fd){
2    float c; *sd = avail;                /* start of D */
▶    while (sa <= fa && sb <= fb)
▶      switch (COMPARE(P[sa].expon, P[sb].expon)) {
5      case -1: COPY(sb); sb++; break;    /* B larger */
6      case  1: COPY(sa); sa++; break;    /* A larger */
7      case  0: c = P[sa].coef + P[sb].coef;
8        if (c) attach(c, P[sa].expon); /* skip 0 */
9        sa++; sb++;
10     }
11   for (; sa <= fa; sa++) COPY(sa);     /* rest of A */
12   for (; sb <= fb; sb++) COPY(sb);     /* rest of B */
13   *fd = avail - 1;
14 }
```

VARIABLES

sa = 0 | sb = 3 | avail = 6

padd: A + B, Term by Term



c = 4 + (3) = **7**.

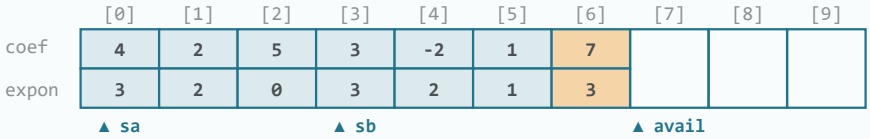
Step 3 / 14

```
1  void padd(int sa,int fa,int sb,int fb,int*sd,int*fd){
2    float c; *sd = avail;              /* start of D */
3    while (sa <= fa && sb <= fb)
4      switch (COMPARE(P[sa].expon, P[sb].expon)) {
5      case -1: COPY(sb); sb++; break;   /* B larger */
6      case  1: COPY(sa); sa++; break;   /* A larger */
▶      case  0: c = P[sa].coef + P[sb].coef;
8        if (c) attach(c, P[sa].expon); /* skip 0 */
9        sa++; sb++;
10     }
11   for (; sa <= fa; sa++) COPY(sa);    /* rest of A */
12   for (; sb <= fb; sb++) COPY(sb);    /* rest of B */
13   *fd = avail - 1;
14 }
```

VARIABLES

sa = 0 | sb = 3 | avail = 6 | c = 7

padd: A + B, Term by Term



c ≠ 0 → `attach(7, 3)` : P[6] = $7x^3$.

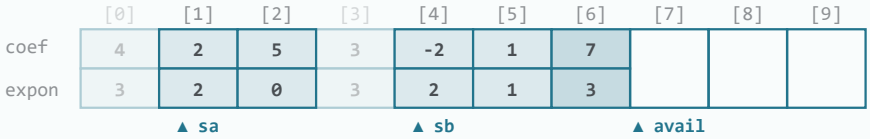
Step 4 / 14

```
1  void padd(int sa,int fa,int sb,int fb,int*sd,int*fd){
2    float c; *sd = avail;                /* start of D */
3    while (sa <= fa && sb <= fb)
4      switch (COMPARE(P[sa].expon, P[sb].expon)) {
5      case -1: COPY(sb); sb++; break;    /* B larger */
6      case  1: COPY(sa); sa++; break;    /* A larger */
7      case  0: c = P[sa].coef + P[sb].coef;
▶        if (c) attach(c, P[sa].expon); /* skip 0 */
9        sa++; sb++;
10     }
11   for (; sa <= fa; sa++) COPY(sa);     /* rest of A */
12   for (; sb <= fb; sb++) COPY(sb);     /* rest of B */
13   *fd = avail - 1;
14 }
```

VARIABLES

`sa = 0` `sb = 3` `avail = 7` `c = 7`

padd: A + B, Term by Term



Both pointers advance.

Step 5 / 14

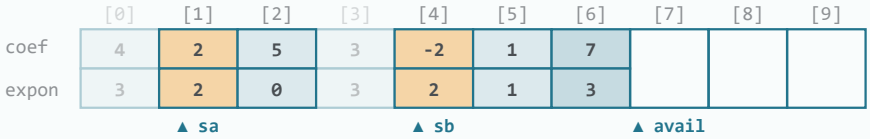
```
1  void padd(int sa,int fa,int sb,int fb,int*sd,int*fd){
2    float c; *sd = avail;                /* start of D */
3    while (sa <= fa && sb <= fb)
4      switch (COMPARE(P[sa].expon, P[sb].expon)) {
5      case -1: COPY(sb); sb++; break;    /* B larger */
6      case  1: COPY(sa); sa++; break;    /* A larger */
7      case  0: c = P[sa].coef + P[sb].coef;
8        if (c) attach(c, P[sa].expon); /* skip 0 */
▶        sa++; sb++;
10     }
11   for (; sa <= fa; sa++) COPY(sa);     /* rest of A */
12   for (; sb <= fb; sb++) COPY(sb);     /* rest of B */
13   *fd = avail - 1;
14 }
```

VARIABLES

sa = 1 sb = 4 avail = 7

c = 7

padd: A + B, Term by Term



COMPARE(2, 2) = **0**: equal exponents.

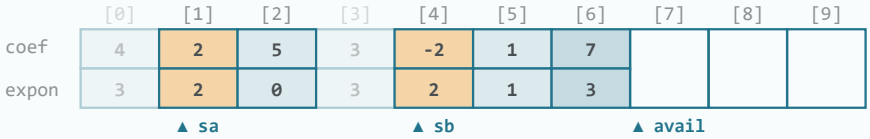
Step 6 / 14

```
1  void padd(int sa,int fa,int sb,int fb,int*sd,int*fd){
2    float c; *sd = avail;              /* start of D */
▶    while (sa <= fa && sb <= fb)
▶      switch (COMPARE(P[sa].expon, P[sb].expon)) {
5      case -1: COPY(sb); sb++; break;   /* B larger */
6      case  1: COPY(sa); sa++; break;   /* A larger */
7      case  0: c = P[sa].coef + P[sb].coef;
8        if (c) attach(c, P[sa].expon); /* skip 0 */
9        sa++; sb++;
10     }
11   for (; sa <= fa; sa++) COPY(sa);    /* rest of A */
12   for (; sb <= fb; sb++) COPY(sb);    /* rest of B */
13   *fd = avail - 1;
14 }
```

VARIABLES

sa = 1 | sb = 4 | avail = 7

padd: A + B, Term by Term



c = 2 + (-2) = **0**.

Step 7 / 14

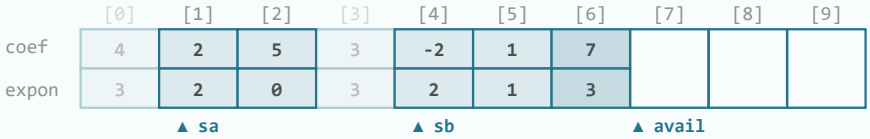
```
1  void padd(int sa,int fa,int sb,int fb,int*sd,int*fd){
2    float c; *sd = avail;                /* start of D */
3    while (sa <= fa && sb <= fb)
4      switch (COMPARE(P[sa].expon, P[sb].expon)) {
5      case -1: COPY(sb); sb++; break;    /* B larger */
6      case  1: COPY(sa); sa++; break;    /* A larger */
▶      case  0: c = P[sa].coef + P[sb].coef;
8        if (c) attach(c, P[sa].expon); /* skip 0 */
9        sa++; sb++;
10     }
11   for (; sa <= fa; sa++) COPY(sa);     /* rest of A */
12   for (; sb <= fb; sb++) COPY(sb);     /* rest of B */
13   *fd = avail - 1;
14 }
```

VARIABLES

sa = 1 sb = 4 avail = 7

c = 0

padd: A + B, Term by Term



c = 0 → the term is **not appended** (the x^2 terms cancel).

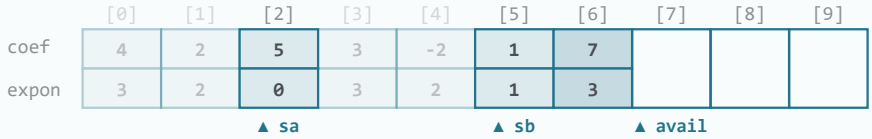
Step 8 / 14

```
1  void padd(int sa,int fa,int sb,int fb,int*sd,int*fd){
2    float c; *sd = avail;                /* start of D */
3    while (sa <= fa && sb <= fb)
4      switch (COMPARE(P[sa].expon, P[sb].expon)) {
5      case -1: COPY(sb); sb++; break;    /* B larger */
6      case  1: COPY(sa); sa++; break;    /* A larger */
7      case  0: c = P[sa].coef + P[sb].coef;
▶        if (c) attach(c, P[sa].expon); /* skip 0 */
9        sa++; sb++;
10     }
11   for (; sa <= fa; sa++) COPY(sa);     /* rest of A */
12   for (; sb <= fb; sb++) COPY(sb);     /* rest of B */
13   *fd = avail - 1;
14 }
```

VARIABLES

`sa = 1` `sb = 4` `avail = 7` `c = 0`

padd: A + B, Term by Term



Both pointers advance.

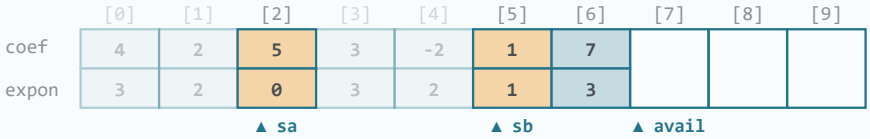
Step 9 / 14

```
1  void padd(int sa,int fa,int sb,int fb,int*sd,int*fd){
2    float c; *sd = avail;                /* start of D */
3    while (sa <= fa && sb <= fb)
4      switch (COMPARE(P[sa].expon, P[sb].expon)) {
5      case -1: COPY(sb); sb++; break;    /* B larger */
6      case  1: COPY(sa); sa++; break;    /* A larger */
7      case  0: c = P[sa].coef + P[sb].coef;
8        if (c) attach(c, P[sa].expon); /* skip 0 */
▶        sa++; sb++;
10     }
11   for (; sa <= fa; sa++) COPY(sa);     /* rest of A */
12   for (; sb <= fb; sb++) COPY(sb);     /* rest of B */
13   *fd = avail - 1;
14 }
```

VARIABLES

sa = 2 sb = 5 avail = 7 c = 0

padd: A + B, Term by Term



COMPARE(0, 1) = **-1**: B's exponent is larger.

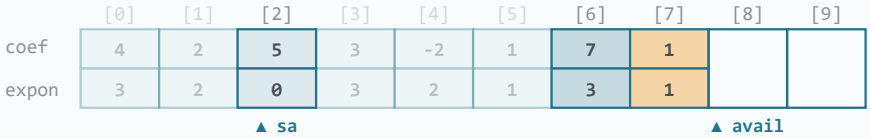
Step 10 / 14

```
1  void padd(int sa,int fa,int sb,int fb,int*sd,int*fd){
2    float c; *sd = avail;              /* start of D */
▶    while (sa <= fa && sb <= fb)
▶      switch (COMPARE(P[sa].expon, P[sb].expon)) {
5      case -1: COPY(sb); sb++; break;   /* B larger */
6      case  1: COPY(sa); sa++; break;   /* A larger */
7      case  0: c = P[sa].coef + P[sb].coef;
8        if (c) attach(c, P[sa].expon); /* skip 0 */
9        sa++; sb++;
10     }
11   for (; sa <= fa; sa++) COPY(sa);    /* rest of A */
12   for (; sb <= fb; sb++) COPY(sb);    /* rest of B */
13   *fd = avail - 1;
14 }
```

VARIABLES

`sa = 2` `sb = 5` `avail = 7`

padd: A + B, Term by Term



B's term (x) is appended to D; **only sb** advances.

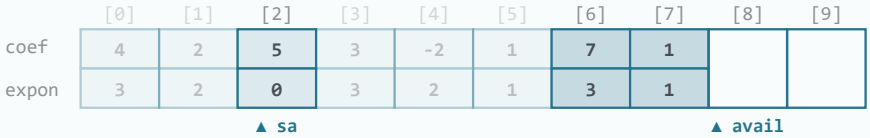
Step 11 / 14

```
1  void padd(int sa,int fa,int sb,int fb,int*sd,int*fd){
2    float c; *sd = avail;              /* start of D */
3    while (sa <= fa && sb <= fb)
4      switch (COMPARE(P[sa].expon, P[sb].expon)) {
▶      case -1: COPY(sb); sb++; break;   /* B larger */
6      case  1: COPY(sa); sa++; break;   /* A larger */
7      case  0: c = P[sa].coef + P[sb].coef;
8        if (c) attach(c, P[sa].expon); /* skip 0 */
9        sa++; sb++;
10     }
11   for (; sa <= fa; sa++) COPY(sa);    /* rest of A */
12   for (; sb <= fb; sb++) COPY(sb);    /* rest of B */
13   *fd = avail - 1;
14 }
```

VARIABLES

sa = 2 sb = 6 avail = 8

padd: A + B, Term by Term



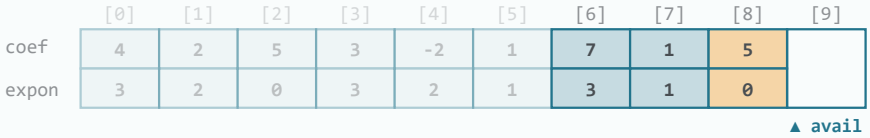
sb = 6 > fb = 5 → B is done; leave the loop.


```
1  void padd(int sa,int fa,int sb,int fb,int*sd,int*fd){
2    float c; *sd = avail;              /* start of D */
▶    while (sa <= fa && sb <= fb)
4      switch (COMPARE(P[sa].expon, P[sb].expon)) {
5      case -1: COPY(sb); sb++; break;   /* B larger */
6      case  1: COPY(sa); sa++; break;   /* A larger */
7      case  0: c = P[sa].coef + P[sb].coef;
8        if (c) attach(c, P[sa].expon); /* skip 0 */
9        sa++; sb++;
10     }
11   for (; sa <= fa; sa++) COPY(sa);    /* rest of A */
12   for (; sb <= fb; sb++) COPY(sb);    /* rest of B */
13   *fd = avail - 1;
14 }
```

VARIABLES

`sa = 2` `sb = 6` `avail = 8`

padd: A + B, Term by Term



The 5 left in A is copied as is.

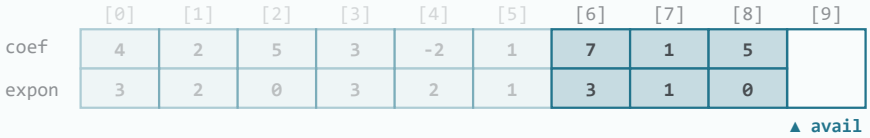
Step 13 / 14

```
1  void padd(int sa,int fa,int sb,int fb,int*sd,int*fd){
2    float c; *sd = avail;              /* start of D */
3    while (sa <= fa && sb <= fb)
4      switch (COMPARE(P[sa].expon, P[sb].expon)) {
5      case -1: COPY(sb); sb++; break;   /* B larger */
6      case  1: COPY(sa); sa++; break;   /* A larger */
7      case  0: c = P[sa].coef + P[sb].coef;
8        if (c) attach(c, P[sa].expon); /* skip 0 */
9        sa++; sb++;
10     }
▶   for (; sa <= fa; sa++) COPY(sa);    /* rest of A */
12   for (; sb <= fb; sb++) COPY(sb);    /* rest of B */
13   *fd = avail - 1;
14 }
```

VARIABLES

sa = 3 sb = 6 avail = 9

padd: A + B, Term by Term



***fd = 8**. D(x) = P[6..8] = **$7x^3 + x + 5$**.

Step 14 / 14

```
1  void padd(int sa,int fa,int sb,int fb,int*sd,int*fd){
2    float c; *sd = avail;                /* start of D */
3    while (sa <= fa && sb <= fb)
4      switch (COMPARE(P[sa].expon, P[sb].expon)) {
5      case -1: COPY(sb); sb++; break;    /* B larger */
6      case  1: COPY(sa); sa++; break;    /* A larger */
7      case  0: c = P[sa].coef + P[sb].coef;
8        if (c) attach(c, P[sa].expon); /* skip 0 */
9        sa++; sb++;
10     }
11   for (; sa <= fa; sa++) COPY(sa);     /* rest of A */
12   for (; sb <= fb; sb++) COPY(sb);     /* rest of B */
▶    *fd = avail - 1;
14 }
```

VARIABLES

sa = 3 | sb = 6 | avail = 9 | sd = 6 | fd = 8

The Complexity of padd

- n terms in A, m in B: every loop round advances **at least one** pointer → at most n + m rounds.
- The copy loops for the leftovers add at most n + m in total → **O(n + m)**.
- Best case: all exponents shared (n = m); every round is case 0 and both pointers move → Ω(n).
- Without sorting, every term would need a search/insert (with shifting) in the result → about **$O(n^2)$**.

Instructor *"If O(n+m) is asked and you can do better, all the better!"*

The price of a shared P array

Results are always appended at `avail`; the space of dead polynomials is never reclaimed. When P fills up we need **compaction** — or a linked list.

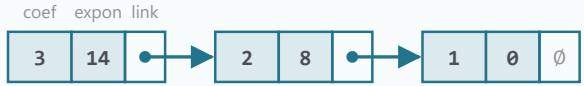
HOMEWORK 2: Pmult, Adding and Multiplying n Polynomials

1. Write a routine `pmult` that multiplies two polynomials (in the P array representation) and analyze its performance.
2. How would you add n polynomials? Design the data structure.
3. How would you multiply n polynomials? Design the data structure.

▶ **Solution**

Storing a Polynomial as a Linked List

```
typedef struct poly_node *poly_pointer;
typedef struct poly_node {
  int coef;
  int expon;
  poly_pointer link;
} poly_node;
```



$$a = 3x^{14} + 2x^{8} + 1$$

- Only the nonzero terms, in **decreasing** exponent order.
- The number of terms need not be known in advance; no shared array, no `avail`, no compaction.
- Inserting/deleting in the middle is **O(1)** by changing pointers (once the place is found).

Addition is still a merge

The list padd also walks with two pointers: **O(n + m)**; result nodes come from `malloc` and must be `free`d when done (Horowitz 4.4).

Helpers: new_term and delete_next_term

```
/* make a term; link it after prev if given */
poly_pointer new_term(int expon, int coef,
                      poly_pointer prev) {
  poly_pointer p = malloc(sizeof *p);
  if (!p) { printf("Out of memory!\n"); exit(1); }
  p->expon = expon;  p->coef = coef;
  if (prev) {                /* middle or end */
    p->link = prev->link;    /* new node first */
    prev->link = p;          /* then prev */
  } else p->link = NULL;     /* stands alone */
  return p;
}
```

```
/* delete the node after prev */
void delete_next_term(
        poly_pointer prev) {
  poly_pointer victim =
        prev->link;
  if (!victim) return;
  prev->link =
        victim->link;
  free(victim);
}
```

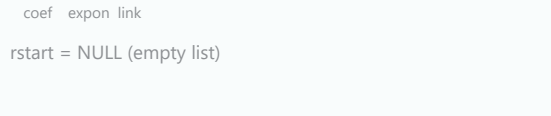
The address is kept in `victim` first so it can be `free`d after the link is updated; “the leftover in memory must be cleaned up”.

Multiplying Two Polynomials Given as Lists

```
poly_pointer place_term(poly_pointer r,
                        int expon, int coef) {
  while (r->link && r->link->expon > expon) r = r->link;
  if (!r->link || r->link->expon < expon)
    new_term(expon, coef, r);          /* new exponent */
  else if ((r->link->coef += coef) == 0)
    delete_next_term(r);               /* became 0 */
  return r;
}
```

- Every term of A is multiplied by every term of B: **exponents add, coefficients multiply**.
- The result list must stay sorted; if a term with the same exponent exists its coefficient is **updated**, and if it becomes 0 the term is **deleted**.
- `start` : the products of p's next term cannot begin **before** the product of the previous one with `qstart` → the search starts there.
- For readability the inner loop body was moved into `place_term` .

poly_mult: (x + 1)·(x − 1)



We multiply A = x + 1 by B = x − 1 (expected $x^2 - 1$). The result list is empty.

Step 1 / 14

```
 1 poly_pointer poly_mult(poly_pointer pstart,
 2                        poly_pointer qstart) {
 3   poly_pointer p, q, r = NULL, start = NULL, rstart=NULL;
 4   for (p = pstart; p; p = p->link) {
 5     r = start;              /* search from here */
 6     for (q = qstart; q; q = q->link) {
 7       int expon = p->expon + q->expon;
 8       int coef = p->coef * q->coef;
 9       if (rstart) r = place_term(r, expon, coef);
10       else rstart=r=start=new_term(expon, coef, NULL);
11       if (q == qstart) start = r;  /* for next p */
12     }
13   }
14   return rstart;
15 }
```

VARIABLES

`p = ?` `q = ?` `r = NULL` `start = NULL` `rstart = NULL`

poly_mult: (x + 1)·(x − 1)

coef expon link

rstart = NULL (empty list)

p: x q: —

p = x. **r = start** = NULL (no list yet).

Step 2 / 14

```
1  poly_pointer poly_mult(poly_pointer pstart,
2                         poly_pointer qstart) {
3    poly_pointer p, q, r = NULL, start = NULL, rstart=NULL;
▶    for (p = pstart; p; p = p->link) {
▶      r = start;              /* search from here */
6      for (q = qstart; q; q = q->link) {
7        int expon = p->expon + q->expon;
8        int coef = p->coef * q->coef;
9        if (rstart) r = place_term(r, expon, coef);
10       else rstart=r=start=new_term(expon, coef, NULL);
11       if (q == qstart) start = r;  /* for next p */
12     }
13   }
14   return rstart;
15 }
```

VARIABLES

p = &(1,1) r = NULL

start = NULL rstart = NULL

poly_mult: (x + 1)·(x − 1)



expon = 1 + 1 = **2**, coef = 1·(1) = **1**.

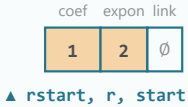
Step 3 / 14

```
1  poly_pointer poly_mult(poly_pointer pstart,
2                         poly_pointer qstart) {
3    poly_pointer p, q, r = NULL, start = NULL, rstart=NULL;
4    for (p = pstart; p; p = p->link) {
5      r = start;              /* search from here */
6      for (q = qstart; q; q = q->link) {
▶        int expon = p->expon + q->expon;
▶        int coef = p->coef * q->coef;
9        if (rstart) r = place_term(r, expon, coef);
10       else rstart=r=start=new_term(expon, coef, NULL);
11       if (q == qstart) start = r;  /* for next p */
12     }
13   }
14   return rstart;
15 }
```

VARIABLES

`p = &(1,1)` `q = &(1,1)` `expon = 2` `coef = 1` `r = NULL` `start = NULL` `rstart = NULL`

poly_mult: (x + 1)·(x − 1)



p: x q: x

The list is empty → the first node is x^2; **rstart = r = start**.

Step 4 / 14

```
 1 poly_pointer poly_mult(poly_pointer pstart,
 2                        poly_pointer qstart) {
 3   poly_pointer p, q, r = NULL, start = NULL, rstart=NULL;
 4   for (p = pstart; p; p = p->link) {
 5     r = start;              /* search from here */
 6     for (q = qstart; q; q = q->link) {
 7       int expon = p->expon + q->expon;
 8       int coef = p->coef * q->coef;
 9       if (rstart) r = place_term(r, expon, coef);
▶        else rstart=r=start=new_term(expon, coef, NULL);
11       if (q == qstart) start = r;  /* for next p */
12     }
13   }
14   return rstart;
15 }
```

VARIABLES

`p = &(1,1)` `q = &(1,1)` `expon = 2` `coef = 1` `r = &(1,2)` `start = &(1,2)` `rstart = &(1,2)`

poly_mult: (x + 1)·(x − 1)



p: x q: x

q was the first term → **start = r** = &(1,2): p's next term will begin its search here.

Step 5 / 14

```
 1 poly_pointer poly_mult(poly_pointer pstart,
 2                        poly_pointer qstart) {
 3   poly_pointer p, q, r = NULL, start = NULL, rstart=NULL;
 4   for (p = pstart; p; p = p->link) {
 5     r = start;              /* search from here */
 6     for (q = qstart; q; q = q->link) {
 7       int expon = p->expon + q->expon;
 8       int coef = p->coef * q->coef;
 9       if (rstart) r = place_term(r, expon, coef);
10       else rstart=r=start=new_term(expon, coef, NULL);
▶        if (q == qstart) start = r;  /* for next p */
12     }
13   }
14   return rstart;
15 }
```

VARIABLES

`p = &(1,1)` `q = &(1,1)` `expon = 2` `coef = 1` `r = &(1,2)` `start = &(1,2)` `rstart = &(1,2)`

poly_mult: (x + 1)·(x − 1)



p: x q: −1

expon = 1 + 0 = **1**, coef = 1·(-1) = **-1**.

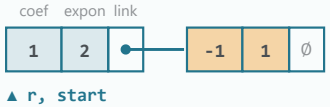
Step 6 / 14

```
1  poly_pointer poly_mult(poly_pointer pstart,
2                         poly_pointer qstart) {
3    poly_pointer p, q, r = NULL, start = NULL, rstart=NULL;
4    for (p = pstart; p; p = p->link) {
5      r = start;              /* search from here */
6      for (q = qstart; q; q = q->link) {
▶        int expon = p->expon + q->expon;
▶        int coef = p->coef * q->coef;
9        if (rstart) r = place_term(r, expon, coef);
10       else rstart=r=start=new_term(expon, coef, NULL);
11       if (q == qstart) start = r;  /* for next p */
12     }
13   }
14   return rstart;
15 }
```

VARIABLES

p = &(1,1) | q = &(-1,0) | expon = 1 | coef = -1 | r = &(1,2) | start = &(1,2) | rstart = &(1,2)

poly_mult: (x + 1)·(x − 1)



p: x q: −1

`place_term` : no term with exponent 1 after r → a **new node** −x is linked after r.

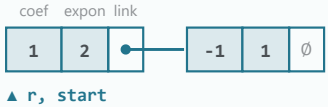
Step 7 / 14

```
 1 poly_pointer poly_mult(poly_pointer pstart,
 2                        poly_pointer qstart) {
 3   poly_pointer p, q, r = NULL, start = NULL, rstart=NULL;
 4   for (p = pstart; p; p = p->link) {
 5     r = start;              /* search from here */
 6     for (q = qstart; q; q = q->link) {
 7       int expon = p->expon + q->expon;
 8       int coef = p->coef * q->coef;
 ▶       if (rstart) r = place_term(r, expon, coef);
10       else rstart=r=start=new_term(expon, coef, NULL);
11       if (q == qstart) start = r;  /* for next p */
12     }
13   }
14   return rstart;
15 }
```

VARIABLES

`p = &(1,1)` `q = &(-1,0)`
`expon = 1` `coef = -1` `r = &(1,2)`
`start = &(1,2)` `rstart = &(1,2)`

poly_mult: (x + 1)·(x − 1)



p: 1 q: —

p = 1. **r = start**: the search begins at node &(1,2), not at the head.

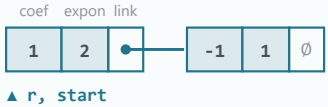
Step 8 / 14

```
 1 poly_pointer poly_mult(poly_pointer pstart,
 2                        poly_pointer qstart) {
 3   poly_pointer p, q, r = NULL, start = NULL, rstart=NULL;
▶  for (p = pstart; p; p = p->link) {
▶    r = start;              /* search from here */
 6    for (q = qstart; q; q = q->link) {
 7      int expon = p->expon + q->expon;
 8      int coef = p->coef * q->coef;
 9      if (rstart) r = place_term(r, expon, coef);
10      else rstart=r=start=new_term(expon, coef, NULL);
11      if (q == qstart) start = r;  /* for next p */
12    }
13  }
14  return rstart;
15 }
```

VARIABLES

p = &(1,0) · r = &(1,2) · start = &(1,2) · rstart = &(1,2)

poly_mult: (x + 1)·(x − 1)



p: 1 q: x

expon = 0 + 1 = **1**, coef = 1·(1) = **1**.

Step 9 / 14

```
 1 poly_pointer poly_mult(poly_pointer pstart,
 2                        poly_pointer qstart) {
 3   poly_pointer p, q, r = NULL, start = NULL, rstart=NULL;
 4   for (p = pstart; p; p = p->link) {
 5     r = start;              /* search from here */
 6     for (q = qstart; q; q = q->link) {
▶        int expon = p->expon + q->expon;
▶        int coef = p->coef * q->coef;
 9       if (rstart) r = place_term(r, expon, coef);
10       else rstart=r=start=new_term(expon, coef, NULL);
11       if (q == qstart) start = r;  /* for next p */
12     }
13   }
14   return rstart;
15 }
```

VARIABLES

`p = &(1,0)` `q = &(1,1)` `expon = 1` `coef = 1` `r = &(1,2)` `start = &(1,2)` `rstart = &(1,2)`

poly_mult: (x + 1)·(x − 1)



p: 1 q: x

`place_term` : a term with exponent 1 exists; its coefficient −1 + 1 = **0** → deleted with `delete_next_term(r)` .

Step 10 / 14

```
 1 poly_pointer poly_mult(poly_pointer pstart,
 2                        poly_pointer qstart) {
 3   poly_pointer p, q, r = NULL, start = NULL, rstart=NULL;
 4   for (p = pstart; p; p = p->link) {
 5     r = start;              /* search from here */
 6     for (q = qstart; q; q = q->link) {
 7       int expon = p->expon + q->expon;
 8       int coef = p->coef * q->coef;
▶        if (rstart) r = place_term(r, expon, coef);
10       else rstart=r=start=new_term(expon, coef, NULL);
11       if (q == qstart) start = r;  /* for next p */
12     }
13   }
14   return rstart;
15 }
```

VARIABLES

`p = &(1,0)` `q = &(1,1)` `expon = 1` `coef = 1` `r = &(1,2)` `start = &(1,2)` `rstart = &(1,2)`

poly_mult: (x + 1)·(x − 1)



p: 1 q: x

q was the first term → **start = r** = &(1,2): p's next term will begin its search here.

Step 11 / 14

```
1  poly_pointer poly_mult(poly_pointer pstart,
2                         poly_pointer qstart) {
3    poly_pointer p, q, r = NULL, start = NULL, rstart=NULL;
4    for (p = pstart; p; p = p->link) {
5      r = start;              /* search from here */
6      for (q = qstart; q; q = q->link) {
7        int expon = p->expon + q->expon;
8        int coef = p->coef * q->coef;
9        if (rstart) r = place_term(r, expon, coef);
10       else rstart=r=start=new_term(expon, coef, NULL);
▶        if (q == qstart) start = r;  /* for next p */
12     }
13   }
14   return rstart;
15 }
```

VARIABLES

- p = &(1,0)
- q = &(1,1)
- expon = 1
- coef = 1
- r = &(1,2)
- start = &(1,2)
- rstart = &(1,2)

poly_mult: (x + 1)·(x − 1)



p: 1 q: −1

expon = 0 + 0 = **0**, coef = 1·(-1) = **-1**.

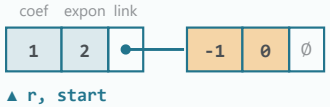
Step 12 / 14

```
 1 poly_pointer poly_mult(poly_pointer pstart,
 2                        poly_pointer qstart) {
 3   poly_pointer p, q, r = NULL, start = NULL, rstart=NULL;
 4   for (p = pstart; p; p = p->link) {
 5     r = start;              /* search from here */
 6     for (q = qstart; q; q = q->link) {
▶        int expon = p->expon + q->expon;
▶        int coef = p->coef * q->coef;
 9       if (rstart) r = place_term(r, expon, coef);
10       else rstart=r=start=new_term(expon, coef, NULL);
11       if (q == qstart) start = r;  /* for next p */
12     }
13   }
14   return rstart;
15 }
```

VARIABLES

`p = &(1,0)` `q = &(-1,0)` `expon = 0` `coef = -1` `r = &(1,2)` `start = &(1,2)` `rstart = &(1,2)`

poly_mult: (x + 1)·(x − 1)



p: 1 q: −1

`place_term` : no term with exponent 0 after r → a **new node** −1 is linked after r.

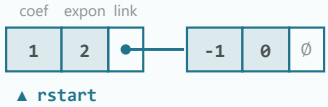
Step 13 / 14

```
 1 poly_pointer poly_mult(poly_pointer pstart,
 2                        poly_pointer qstart) {
 3   poly_pointer p, q, r = NULL, start = NULL, rstart=NULL;
 4   for (p = pstart; p; p = p->link) {
 5     r = start;              /* search from here */
 6     for (q = qstart; q; q = q->link) {
 7       int expon = p->expon + q->expon;
 8       int coef = p->coef * q->coef;
▶9       if (rstart) r = place_term(r, expon, coef);
10       else rstart=r=start=new_term(expon, coef, NULL);
11       if (q == qstart) start = r;  /* for next p */
12     }
13   }
14   return rstart;
15 }
```

VARIABLES

`p = &(1,0)` `q = &(-1,0)`
`expon = 0` `coef = -1` `r = &(1,2)`
`start = &(1,2)` `rstart = &(1,2)`

poly_mult: (x + 1)·(x − 1)



$(x + 1)(x - 1) = x^2 - 1$

Result: **$x^2 - 1$**. n·m = 4 products; −x and +x summed to 0 and were deleted.

Step 14 / 14

```
1  poly_pointer poly_mult(poly_pointer pstart,
2                         poly_pointer qstart) {
3    poly_pointer p, q, r = NULL, start = NULL, rstart=NULL;
4    for (p = pstart; p; p = p->link) {
5      r = start;              /* search from here */
6      for (q = qstart; q; q = q->link) {
7        int expon = p->expon + q->expon;
8        int coef = p->coef * q->coef;
9        if (rstart) r = place_term(r, expon, coef);
10       else rstart=r=start=new_term(expon, coef, NULL);
11       if (q == qstart) start = r;  /* for next p */
12     }
13   }
▶    return rstart;
15 }
```

VARIABLES

`rstart = &(1,2)`

Other Solutions to the Problem (A: n Terms, B: m Terms)

	method	complexity
1	Multiply everything, append each product at the end; then sort and combine	$O(n^2m^2)$
2	Multiply everything; scan from the head each time and insert in place	$O(n^2m^2)$
3	Build a list for the product with each term of A and merge it into the result at each step	$O(n^2m)$
4	Method 3 with the smaller polynomial outside	$O(\min(n,m)^2 \cdot \max(n,m))$
5	A separate list for each term of A; merge them all (in pairs) at the end	$O(m \cdot n \cdot \lg n)$
6	Method 5 with the smaller polynomial outside	$O(m \cdot n \cdot \lg \min(m,n))$
7	Use an array whose n-th index holds the n-th exponent	—

Instructor *"An array whose n-th index holds the n-th exponent? NEVER!"*

A Sparse Matrix: Most Entries Are Zero

	0	1	2	3	4	5
0	15	0	0	22	0	-15
1	0	11	3	0	0	0
2	0	0	0	-6	0	0
3	0	0	0	0	0	0
4	91	0	0	0	0	0
5	0	0	28	0	0	0

Only 8 of the 36 cells are nonzero

A precise definition is hard; storing it in a 2D array **wastes memory** on many zeros. Solution: store each nonzero entry as a ⟨**row, col, value**⟩ triple.

```
#define MAX_TERMS 101
#define MAX_COL 50   /* max columns */
typedef struct {
  int row, col, value;
} term;
term M[MAX_TERMS];
```

M[0] is the header: number of rows, number of columns, number of nonzero entries. The triples are sorted **by row, then by column**.

The Triple Table, and When It Pays Off

	row	col	value
M[0]	6	6	8
M[1]	0	0	15
M[2]	0	3	22
M[3]	0	5	-15
M[4]	1	1	11
M[5]	1	2	3
M[6]	2	3	-6
M[7]	4	0	91
M[8]	5	2	28

Memory

3·(t + 1) integers instead of rows·cols. They break even at **t ≈ rows·cols/3**, i.e. when a third of the matrix is nonzero. Denser than that, this structure is not used.

In this example

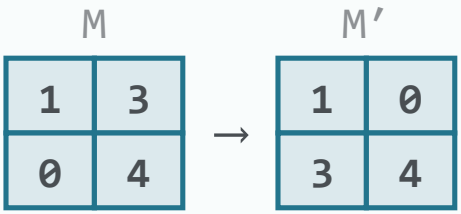
3·9 = 27 integers instead of 36.

ADT operations

Create(max_row, max_col), Transpose(a), Add(a, b), Multiply(a, b) — Horowitz ADT 2.3. Other representations exist too (linked, CSR…).

Transpose: M[i][j] → M'[j][i]

We write the rows as columns:

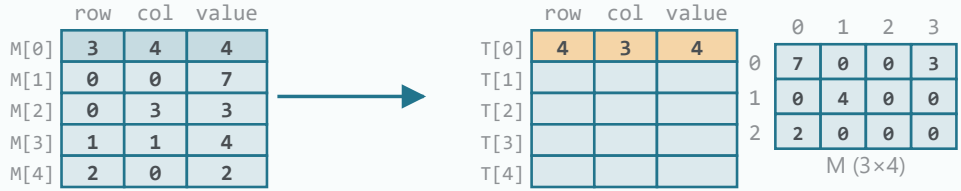


- Simply swapping row and col in the triples is not enough: the result **breaks the row order**.
- Swapping and then sorting, or inserting each entry by **shifting**, is expensive.
- Solution: the rows of the transpose are the columns of M; scan **column by column** and emit them in order.

Instructor *"If you are shifting and moving data around, you are not doing the data structure justice!"*

Starting the inner loop of `transpose` at `j = 0` also scans the header M[0]. (M[0].col = number of columns never equals any i, so the result is fine, but it is wasted work.) It should be `j = 1`.

transpose: Scan Every Term for Each Column



Header: T is 4×3 (rows ↔ columns), still **4 terms**.

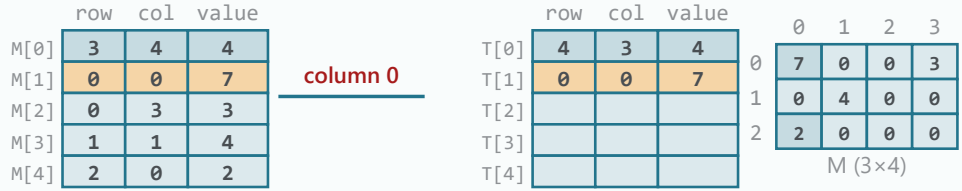
Step 1 / 18

```
1  void transpose(term M[], term T[]) {
2    int n = M[0].value, i, j, cur;
▶    T[0].row = M[0].col;  T[0].col = M[0].row;
▶    T[0].value = n;
5    if (n > 0) {
6      cur = 1;                        /* next in T */
7      for (i = 0; i < M[0].col; i++)     /* each col */
8        for (j = 1; j <= n; j++)         /* each term */
9          if (M[j].col == i) {
10           T[cur].row = M[j].col; T[cur].col = M[j].row;
11           T[cur].value = M[j].value;  cur++;
12         }
13   }
14 }
```

VARIABLES

n = 4

transpose: Scan Every Term for Each Column



Column 0: M[1].col = 0 **matches** → T[1] = (0, 0, 7), cur = 2.

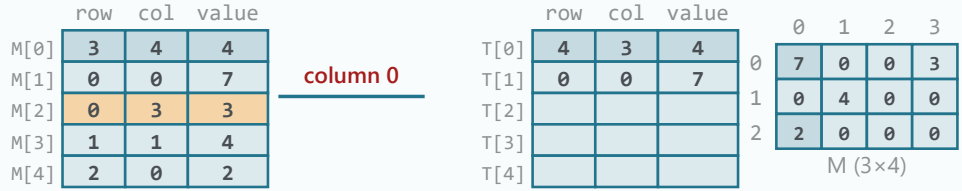
Step 2 / 18

```
1  void transpose(term M[], term T[]) {
2    int n = M[0].value, i, j, cur;
3    T[0].row = M[0].col;  T[0].col = M[0].row;
4    T[0].value = n;
5    if (n > 0) {
6      cur = 1;                          /* next in T */
7      for (i = 0; i < M[0].col; i++)      /* each col */
8        for (j = 1; j <= n; j++)          /* each term */
▶          if (M[j].col == i) {
▶            T[cur].row = M[j].col; T[cur].col = M[j].row;
▶            T[cur].value = M[j].value;  cur++;
12         }
13   }
14 }
```

VARIABLES

`n = 4` `cur = 2` `i = 0` `j = 1` `M[j].col = 0`

transpose: Scan Every Term for Each Column



Column 0: M[2].col = 3 ≠ 0 → skip.

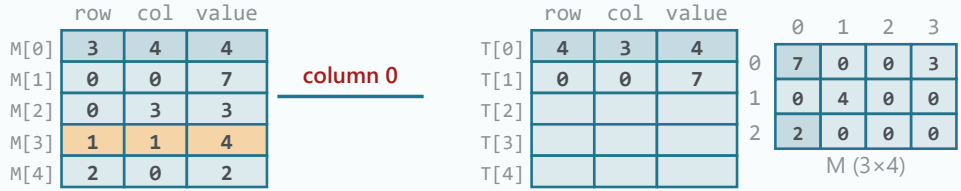
Step 3 / 18

```
1  void transpose(term M[], term T[]) {
2    int n = M[0].value, i, j, cur;
3    T[0].row = M[0].col;  T[0].col = M[0].row;
4    T[0].value = n;
5    if (n > 0) {
6      cur = 1;                          /* next in T */
7      for (i = 0; i < M[0].col; i++)      /* each col */
8        for (j = 1; j <= n; j++)          /* each term */
▶          if (M[j].col == i) {
10           T[cur].row = M[j].col; T[cur].col = M[j].row;
11           T[cur].value = M[j].value;  cur++;
12         }
13   }
14 }
```

VARIABLES

`n = 4` `cur = 2` `i = 0` `j = 2` `M[j].col = 3`

transpose: Scan Every Term for Each Column



Column 0: M[3].col = 1 ≠ 0 → skip.

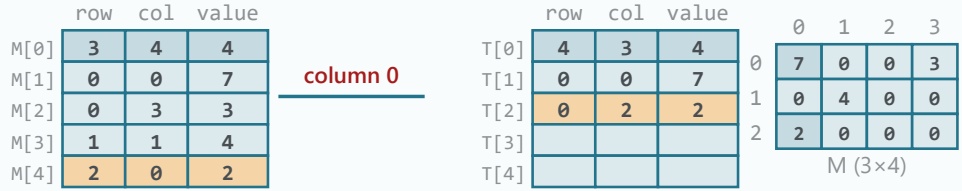
Step 4 / 18

```
1  void transpose(term M[], term T[]) {
2    int n = M[0].value, i, j, cur;
3    T[0].row = M[0].col;  T[0].col = M[0].row;
4    T[0].value = n;
5    if (n > 0) {
6      cur = 1;                          /* next in T */
7      for (i = 0; i < M[0].col; i++)      /* each col */
8        for (j = 1; j <= n; j++)          /* each term */
▶          if (M[j].col == i) {
10           T[cur].row = M[j].col; T[cur].col = M[j].row;
11           T[cur].value = M[j].value;  cur++;
12         }
13   }
14 }
```

VARIABLES

n = 4 | cur = 2 | i = 0 | j = 3 | M[j].col = 1

transpose: Scan Every Term for Each Column



Column 0: M[4].col = 0 **matches** → T[2] = (0, 2, 2), cur = 3.

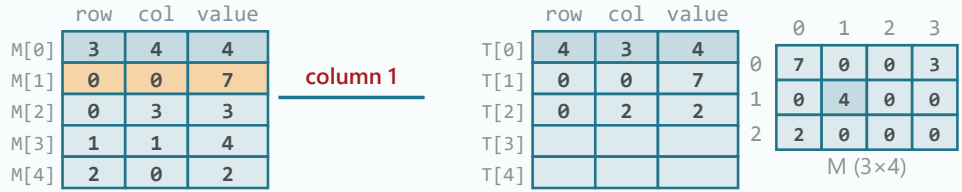
Step 5 / 18

```
1  void transpose(term M[], term T[]) {
2    int n = M[0].value, i, j, cur;
3    T[0].row = M[0].col;  T[0].col = M[0].row;
4    T[0].value = n;
5    if (n > 0) {
6      cur = 1;                          /* next in T */
7      for (i = 0; i < M[0].col; i++)      /* each col */
8        for (j = 1; j <= n; j++)          /* each term */
▶          if (M[j].col == i) {
▶            T[cur].row = M[j].col; T[cur].col = M[j].row;
▶            T[cur].value = M[j].value;  cur++;
12         }
13   }
14 }
```

VARIABLES

n = 4 | cur = 3 | i = 0 | j = 4 | M[j].col = 0

transpose: Scan Every Term for Each Column



Column 1: M[1].col = 0 ≠ 1 → skip.

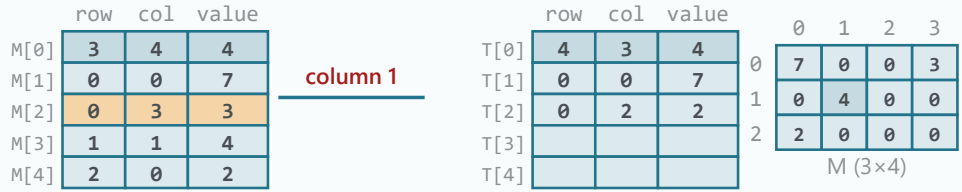
Step 6 / 18

```
1  void transpose(term M[], term T[]) {
2    int n = M[0].value, i, j, cur;
3    T[0].row = M[0].col;  T[0].col = M[0].row;
4    T[0].value = n;
5    if (n > 0) {
6      cur = 1;                          /* next in T */
7      for (i = 0; i < M[0].col; i++)      /* each col */
8        for (j = 1; j <= n; j++)          /* each term */
▶         if (M[j].col == i) {
10          T[cur].row = M[j].col; T[cur].col = M[j].row;
11          T[cur].value = M[j].value;  cur++;
12        }
13   }
14 }
```

VARIABLES

`n = 4` `cur = 3` `i = 1` `j = 1` `M[j].col = 0`

transpose: Scan Every Term for Each Column



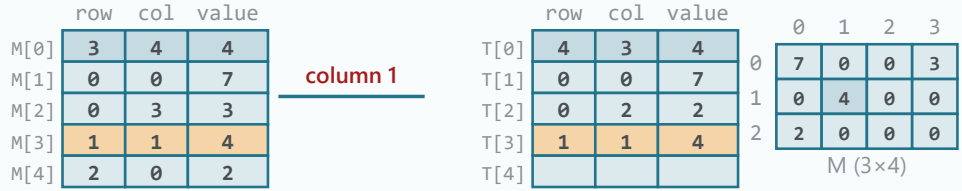
Column 1: M[2].col = 3 ≠ 1 → skip.


```
1  void transpose(term M[], term T[]) {
2    int n = M[0].value, i, j, cur;
3    T[0].row = M[0].col;  T[0].col = M[0].row;
4    T[0].value = n;
5    if (n > 0) {
6      cur = 1;                          /* next in T */
7      for (i = 0; i < M[0].col; i++)      /* each col */
8        for (j = 1; j <= n; j++)          /* each term */
▶          if (M[j].col == i) {
10           T[cur].row = M[j].col; T[cur].col = M[j].row;
11           T[cur].value = M[j].value;  cur++;
12         }
13   }
14 }
```

VARIABLES

n = 4 | cur = 3 | i = 1 | j = 2 | M[j].col = 3

transpose: Scan Every Term for Each Column



Column 1: M[3].col = 1 **matches** → T[3] = (1, 1, 4), cur = 4.

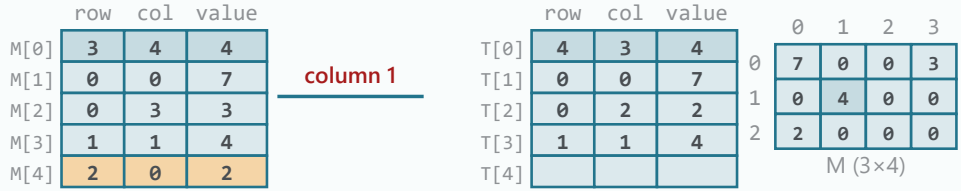
Step 8 / 18

```
1  void transpose(term M[], term T[]) {
2    int n = M[0].value, i, j, cur;
3    T[0].row = M[0].col;  T[0].col = M[0].row;
4    T[0].value = n;
5    if (n > 0) {
6      cur = 1;                          /* next in T */
7      for (i = 0; i < M[0].col; i++)      /* each col */
8        for (j = 1; j <= n; j++)          /* each term */
▶          if (M[j].col == i) {
▶            T[cur].row = M[j].col; T[cur].col = M[j].row;
▶            T[cur].value = M[j].value;  cur++;
12         }
13   }
14 }
```

VARIABLES

n = 4 | cur = 4 | i = 1 | j = 3 | M[j].col = 1

transpose: Scan Every Term for Each Column



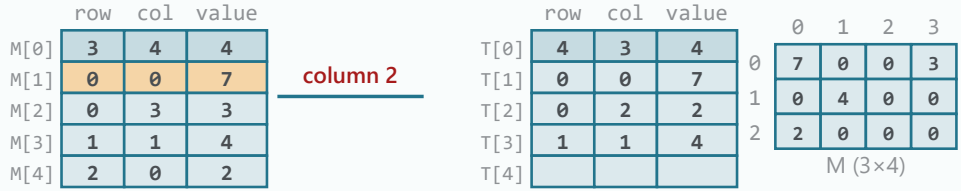
Column 1: M[4].col = 0 ≠ 1 → skip.


```
1  void transpose(term M[], term T[]) {
2    int n = M[0].value, i, j, cur;
3    T[0].row = M[0].col;  T[0].col = M[0].row;
4    T[0].value = n;
5    if (n > 0) {
6      cur = 1;                          /* next in T */
7      for (i = 0; i < M[0].col; i++)      /* each col */
8        for (j = 1; j <= n; j++)          /* each term */
▶          if (M[j].col == i) {
10           T[cur].row = M[j].col; T[cur].col = M[j].row;
11           T[cur].value = M[j].value;  cur++;
12         }
13   }
14 }
```

VARIABLES

`n = 4` `cur = 4` `i = 1` `j = 4` `M[j].col = 0`

transpose: Scan Every Term for Each Column



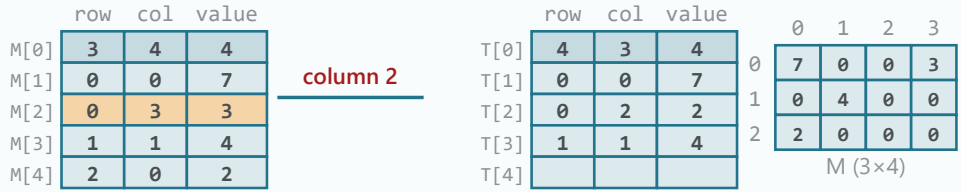
Column 2: M[1].col = 0 ≠ 2 → skip.


```
1  void transpose(term M[], term T[]) {
2    int n = M[0].value, i, j, cur;
3    T[0].row = M[0].col;  T[0].col = M[0].row;
4    T[0].value = n;
5    if (n > 0) {
6      cur = 1;                          /* next in T */
7      for (i = 0; i < M[0].col; i++)      /* each col */
8        for (j = 1; j <= n; j++)          /* each term */
▶          if (M[j].col == i) {
10           T[cur].row = M[j].col; T[cur].col = M[j].row;
11           T[cur].value = M[j].value;  cur++;
12         }
13   }
14 }
```

VARIABLES

`n = 4` `cur = 4` `i = 2` `j = 1` `M[j].col = 0`

transpose: Scan Every Term for Each Column



Column 2: M[2].col = 3 ≠ 2 → skip.

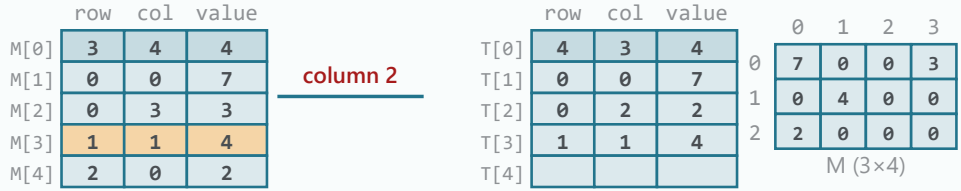
Step 11 / 18

```
1  void transpose(term M[], term T[]) {
2    int n = M[0].value, i, j, cur;
3    T[0].row = M[0].col;  T[0].col = M[0].row;
4    T[0].value = n;
5    if (n > 0) {
6      cur = 1;                          /* next in T */
7      for (i = 0; i < M[0].col; i++)      /* each col */
8        for (j = 1; j <= n; j++)          /* each term */
▶          if (M[j].col == i) {
10           T[cur].row = M[j].col; T[cur].col = M[j].row;
11           T[cur].value = M[j].value;  cur++;
12         }
13   }
14 }
```

VARIABLES

n = 4 | cur = 4 | i = 2 | j = 2 | M[j].col = 3

transpose: Scan Every Term for Each Column



Column 2: M[3].col = 1 ≠ 2 → skip.

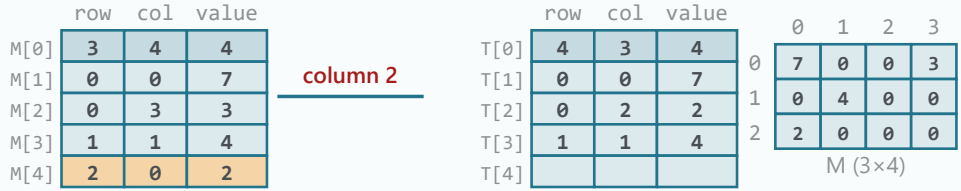
Step 12 / 18

```
1  void transpose(term M[], term T[]) {
2    int n = M[0].value, i, j, cur;
3    T[0].row = M[0].col;  T[0].col = M[0].row;
4    T[0].value = n;
5    if (n > 0) {
6      cur = 1;                          /* next in T */
7      for (i = 0; i < M[0].col; i++)      /* each col */
8        for (j = 1; j <= n; j++)          /* each term */
▶          if (M[j].col == i) {
10           T[cur].row = M[j].col; T[cur].col = M[j].row;
11           T[cur].value = M[j].value;  cur++;
12         }
13   }
14 }
```

VARIABLES

n = 4 | cur = 4 | i = 2 | j = 3 | M[j].col = 1

transpose: Scan Every Term for Each Column



Column 2: M[4].col = 0 ≠ 2 → skip.

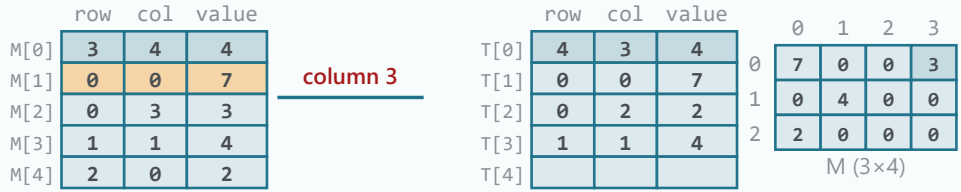
Step 13 / 18

```
1  void transpose(term M[], term T[]) {
2    int n = M[0].value, i, j, cur;
3    T[0].row = M[0].col;  T[0].col = M[0].row;
4    T[0].value = n;
5    if (n > 0) {
6      cur = 1;                          /* next in T */
7      for (i = 0; i < M[0].col; i++)      /* each col */
8        for (j = 1; j <= n; j++)          /* each term */
▶          if (M[j].col == i) {
10           T[cur].row = M[j].col; T[cur].col = M[j].row;
11           T[cur].value = M[j].value;  cur++;
12         }
13   }
14 }
```

VARIABLES

n = 4 | cur = 4 | i = 2 | j = 4 | M[j].col = 0

transpose: Scan Every Term for Each Column



Column 3: M[1].col = 0 ≠ 3 → skip.

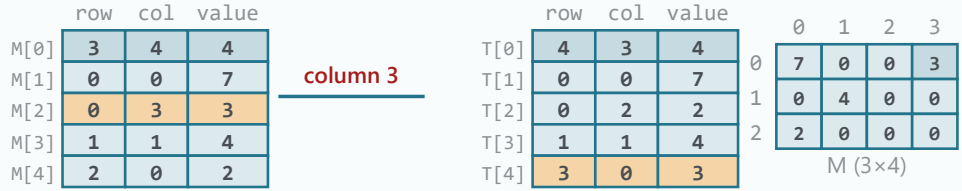
Step 14 / 18

```
1  void transpose(term M[], term T[]) {
2    int n = M[0].value, i, j, cur;
3    T[0].row = M[0].col;  T[0].col = M[0].row;
4    T[0].value = n;
5    if (n > 0) {
6      cur = 1;                          /* next in T */
7      for (i = 0; i < M[0].col; i++)      /* each col */
8        for (j = 1; j <= n; j++)          /* each term */
▶          if (M[j].col == i) {
10           T[cur].row = M[j].col; T[cur].col = M[j].row;
11           T[cur].value = M[j].value;  cur++;
12         }
13   }
14 }
```

VARIABLES

n = 4 | cur = 4 | i = 3 | j = 1 | M[j].col = 0

transpose: Scan Every Term for Each Column



Column 3: M[2].col = 3 **matches** → T[4] = (3, 0, 3), cur = 5.

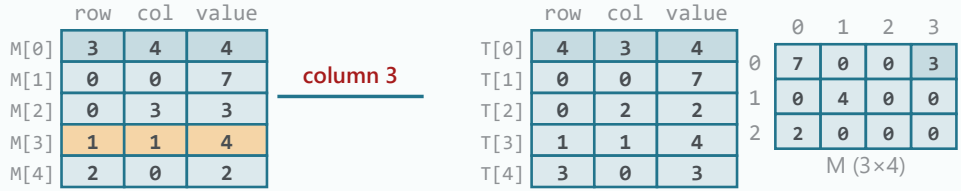
Step 15 / 18

```
1  void transpose(term M[], term T[]) {
2    int n = M[0].value, i, j, cur;
3    T[0].row = M[0].col;  T[0].col = M[0].row;
4    T[0].value = n;
5    if (n > 0) {
6      cur = 1;                          /* next in T */
7      for (i = 0; i < M[0].col; i++)      /* each col */
8        for (j = 1; j <= n; j++)          /* each term */
▶          if (M[j].col == i) {
▶            T[cur].row = M[j].col; T[cur].col = M[j].row;
▶            T[cur].value = M[j].value;  cur++;
12         }
13   }
14 }
```

VARIABLES

n = 4 | cur = 5 | i = 3 | j = 2 | M[j].col = 3

transpose: Scan Every Term for Each Column



Column 3: M[3].col = 1 ≠ 3 → skip.

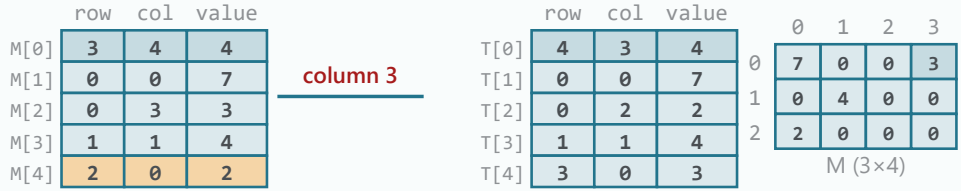
Step 16 / 18

```
1  void transpose(term M[], term T[]) {
2    int n = M[0].value, i, j, cur;
3    T[0].row = M[0].col;  T[0].col = M[0].row;
4    T[0].value = n;
5    if (n > 0) {
6      cur = 1;                          /* next in T */
7      for (i = 0; i < M[0].col; i++)      /* each col */
8        for (j = 1; j <= n; j++)          /* each term */
▶         if (M[j].col == i) {
10          T[cur].row = M[j].col; T[cur].col = M[j].row;
11          T[cur].value = M[j].value;  cur++;
12        }
13   }
14 }
```

VARIABLES

n = 4 | cur = 5 | i = 3 | j = 3 | M[j].col = 1

transpose: Scan Every Term for Each Column



Column 3: M[4].col = 0 ≠ 3 → skip.

Step 17 / 18

```
1  void transpose(term M[], term T[]) {
2    int n = M[0].value, i, j, cur;
3    T[0].row = M[0].col;  T[0].col = M[0].row;
4    T[0].value = n;
5    if (n > 0) {
6      cur = 1;                          /* next in T */
7      for (i = 0; i < M[0].col; i++)      /* each col */
8        for (j = 1; j <= n; j++)          /* each term */
▶          if (M[j].col == i) {
10           T[cur].row = M[j].col; T[cur].col = M[j].row;
11           T[cur].value = M[j].value;  cur++;
12         }
13   }
14 }
```

VARIABLES

n = 4 | cur = 5 | i = 3 | j = 4 | M[j].col = 0

transpose: Scan Every Term for Each Column

	row	col	value
M[0]	3	4	4
M[1]	0	0	7
M[2]	0	3	3
M[3]	1	1	4
M[4]	2	0	2

	row	col	value
T[0]	4	3	4
T[1]	0	0	7
T[2]	0	2	2
T[3]	1	1	4
T[4]	3	0	3

	0	1	2	3
0	7	0	0	3
1	0	4	0	0
2	2	0	0	0

M (3×4)

Done: T is **sorted by row for free** (M's rows were sorted). 4 columns × 4 terms = **16 comparisons**.

Step 18 / 18

```
1  void transpose(term M[], term T[]) {
2    int n = M[0].value, i, j, cur;
3    T[0].row = M[0].col;  T[0].col = M[0].row;
4    T[0].value = n;
5    if (n > 0) {
6      cur = 1;                        /* next in T */
7      for (i = 0; i < M[0].col; i++)     /* each col */
8        for (j = 1; j <= n; j++)         /* each term */
9          if (M[j].col == i) {
10           T[cur].row = M[j].col; T[cur].col = M[j].row;
11           T[cur].value = M[j].value;  cur++;
12         }
13   }
▶ }
```

VARIABLES

`n = 4` `cur = 5`

The Complexity of transpose

- The outer loop runs **cols** times, the inner **terms** times → **O(cols · terms)**.
- For a dense matrix terms = rows·cols → **O(cols² · rows)**: worse than the O(rows·cols) transpose of a 2D array!
- We saved space but lost time — worthwhile only when terms ≪ rows·cols.
- The problem: scanning the same array M **again for every column**.

The way out

If we **knew in advance** where each term goes in T, we could scan M once and put each term straight into place.

Example

If row 0 of the transpose has 2 entries and row 1 has 3: row 0 starts at T[1], row 1 at T[3], row 2 at T[6].

fast_transpose: Count First, Then Place

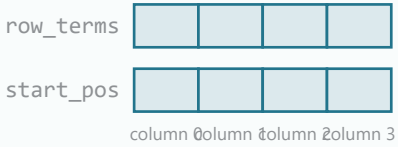
```
void fast_transpose(term M[], term T[]) {
  int i, j, nc = M[0].col, nt = M[0].value;
  int row_terms[MAX_COL], start_pos[MAX_COL];
  T[0].row = nc;  T[0].col = M[0].row;  T[0].value = nt;
  if (nt <= 0) return;                    /* empty matrix */
  for (i = 0; i < nc; i++) row_terms[i] = 0;
  for (i = 1; i <= nt; i++) row_terms[M[i].col]++;
  start_pos[0] = 1;
  for (i = 1; i < nc; i++)
    start_pos[i] = start_pos[i-1] + row_terms[i-1];
  for (i = 1; i <= nt; i++) {
    j = start_pos[M[i].col]++;        /* take slot, bump */
    T[j].row = M[i].col;  T[j].col = M[i].row;
    T[j].value = M[i].value;
  }
}
```

	row	col	value
T[0]	6	6	8
T[1]	0	0	15
T[2]	0	4	91
T[3]	1	1	11
T[4]	2	1	3
T[5]	2	5	28
T[6]	3	0	22
T[7]	3	2	-6
T[8]	5	0	-15

In the 6×6 example **row_terms = [2, 1, 2, 2, 0, 1]**, **start_pos = [1, 3, 4, 6, 8, 8]**.

fast_transpose (1): Count and Find the Starts

	row	col	value
M[0]	3	4	4
M[1]	0	0	7
M[2]	0	3	3
M[3]	1	1	4
M[4]	2	0	2



Copy the header: T is 4×3 with 4 terms. nt > 0 → go on.

Step 1 / 10

```
1  void fast_transpose(term M[], term T[]) {
2    int i, j, nc = M[0].col, nt = M[0].value;
3    int row_terms[MAX_COL], start_pos[MAX_COL];
▶    T[0].row = nc;  T[0].col = M[0].row;
▶    T[0].value = nt;
6    if (nt <= 0) return;                    /* empty */
7    for (i = 0; i < nc; i++) row_terms[i] = 0;
8    for (i = 1; i <= nt; i++) row_terms[M[i].col]++;
9    start_pos[0] = 1;
10   for (i = 1; i < nc; i++)
11     start_pos[i] = start_pos[i-1] + row_terms[i-1];
12   /* phase 2: put the terms in place */
```

VARIABLES

nc = 4 nt = 4

fast_transpose (1): Count and Find the Starts

	row	col	value
M[0]	3	4	4
M[1]	0	0	7
M[2]	0	3	3
M[3]	1	1	4
M[4]	2	0	2

	column 0	column 1	column 2	column 3
row_terms	0	0	0	0
start_pos				

Zero `row_terms` : a counter for each row of T (= each column of M).

Step 2 / 10

```
1  void fast_transpose(term M[], term T[]) {
2    int i, j, nc = M[0].col, nt = M[0].value;
3    int row_terms[MAX_COL], start_pos[MAX_COL];
4    T[0].row = nc;  T[0].col = M[0].row;
5    T[0].value = nt;
6    if (nt <= 0) return;                    /* empty */
▶    for (i = 0; i < nc; i++) row_terms[i] = 0;
8    for (i = 1; i <= nt; i++) row_terms[M[i].col]++;
9    start_pos[0] = 1;
10   for (i = 1; i < nc; i++)
11     start_pos[i] = start_pos[i-1] + row_terms[i-1];
12   /* phase 2: put the terms in place */
```

VARIABLES

`nc = 4` `nt = 4`

fast_transpose (1): Count and Find the Starts

	row	col	value
M[0]	3	4	4
M[1]	0	0	7
M[2]	0	3	3
M[3]	1	1	4
M[4]	2	0	2

	column 0	column 1	column 2	column 3
row_terms	1	0	0	0
start_pos				

M[1] is in column 0 → **row_terms[0]++** = 1.

Step 3 / 10

```
1  void fast_transpose(term M[], term T[]) {
2    int i, j, nc = M[0].col, nt = M[0].value;
3    int row_terms[MAX_COL], start_pos[MAX_COL];
4    T[0].row = nc;  T[0].col = M[0].row;
5    T[0].value = nt;
6    if (nt <= 0) return;                    /* empty */
7    for (i = 0; i < nc; i++) row_terms[i] = 0;
▶    for (i = 1; i <= nt; i++) row_terms[M[i].col]++;
9    start_pos[0] = 1;
10   for (i = 1; i < nc; i++)
11     start_pos[i] = start_pos[i-1] + row_terms[i-1];
12   /* phase 2: put the terms in place */
```

VARIABLES

`nc = 4` `nt = 4` `i = 1` `M[i].col = 0`

fast_transpose (1): Count and Find the Starts

	row	col	value
M[0]	3	4	4
M[1]	0	0	7
M[2]	0	3	3
M[3]	1	1	4
M[4]	2	0	2

	column 0	column 1	column 2	column 3
row_terms	1	0	0	1
start_pos				

M[2] is in column 3 → **row_terms[3]++** = 1.

Step 4 / 10

```
1  void fast_transpose(term M[], term T[]) {
2    int i, j, nc = M[0].col, nt = M[0].value;
3    int row_terms[MAX_COL], start_pos[MAX_COL];
4    T[0].row = nc;  T[0].col = M[0].row;
5    T[0].value = nt;
6    if (nt <= 0) return;                    /* empty */
7    for (i = 0; i < nc; i++) row_terms[i] = 0;
▶    for (i = 1; i <= nt; i++) row_terms[M[i].col]++;
9    start_pos[0] = 1;
10   for (i = 1; i < nc; i++)
11     start_pos[i] = start_pos[i-1] + row_terms[i-1];
12   /* phase 2: put the terms in place */
```

VARIABLES

`nc = 4` `nt = 4` `i = 2` `M[i].col = 3`

fast_transpose (1): Count and Find the Starts

	row	col	value
M[0]	3	4	4
M[1]	0	0	7
M[2]	0	3	3
M[3]	1	1	4
M[4]	2	0	2

	column 0	column 1	column 2	column 3
row_terms	1	1	0	1
start_pos				

M[3] is in column 1 → **row_terms[1]++** = 1.

Step 5 / 10

```
1  void fast_transpose(term M[], term T[]) {
2    int i, j, nc = M[0].col, nt = M[0].value;
3    int row_terms[MAX_COL], start_pos[MAX_COL];
4    T[0].row = nc;  T[0].col = M[0].row;
5    T[0].value = nt;
6    if (nt <= 0) return;                    /* empty */
7    for (i = 0; i < nc; i++) row_terms[i] = 0;
▶    for (i = 1; i <= nt; i++) row_terms[M[i].col]++;
9    start_pos[0] = 1;
10   for (i = 1; i < nc; i++)
11     start_pos[i] = start_pos[i-1] + row_terms[i-1];
12   /* phase 2: put the terms in place */
```

VARIABLES

`nc = 4` `nt = 4` `i = 3` `M[i].col = 1`

fast_transpose (1): Count and Find the Starts

	row	col	value
M[0]	3	4	4
M[1]	0	0	7
M[2]	0	3	3
M[3]	1	1	4
M[4]	2	0	2

	column 0	column 1	column 2	column 3
row_terms	2	1	0	1
start_pos				

M[4] is in column 0 → **row_terms[0]++** = 2.

Step 6 / 10

```
1  void fast_transpose(term M[], term T[]) {
2    int i, j, nc = M[0].col, nt = M[0].value;
3    int row_terms[MAX_COL], start_pos[MAX_COL];
4    T[0].row = nc;  T[0].col = M[0].row;
5    T[0].value = nt;
6    if (nt <= 0) return;                    /* empty */
7    for (i = 0; i < nc; i++) row_terms[i] = 0;
▶    for (i = 1; i <= nt; i++) row_terms[M[i].col]++;
9    start_pos[0] = 1;
10   for (i = 1; i < nc; i++)
11     start_pos[i] = start_pos[i-1] + row_terms[i-1];
12   /* phase 2: put the terms in place */
```

VARIABLES

`nc = 4` `nt = 4` `i = 4` `M[i].col = 0`

fast_transpose (1): Count and Find the Starts

	row	col	value
M[0]	3	4	4
M[1]	0	0	7
M[2]	0	3	3
M[3]	1	1	4
M[4]	2	0	2

	column 0	column 1	column 2	column 3
row_terms	2	1	0	1
start_pos	1			

The terms of column 0 start at T[1]: **start_pos[0] = 1**.

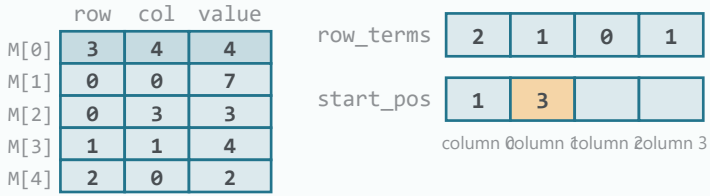
Step 7 / 10

```
1  void fast_transpose(term M[], term T[]) {
2    int i, j, nc = M[0].col, nt = M[0].value;
3    int row_terms[MAX_COL], start_pos[MAX_COL];
4    T[0].row = nc;  T[0].col = M[0].row;
5    T[0].value = nt;
6    if (nt <= 0) return;                    /* empty */
7    for (i = 0; i < nc; i++) row_terms[i] = 0;
8    for (i = 1; i <= nt; i++) row_terms[M[i].col]++;
▶    start_pos[0] = 1;
10   for (i = 1; i < nc; i++)
11     start_pos[i] = start_pos[i-1] + row_terms[i-1];
12   /* phase 2: put the terms in place */
```

VARIABLES

nc = 4 nt = 4

fast_transpose (1): Count and Find the Starts



start_pos[1] = 1 + 2 = **3**.

Step 8 / 10

```
1  void fast_transpose(term M[], term T[]) {
2    int i, j, nc = M[0].col, nt = M[0].value;
3    int row_terms[MAX_COL], start_pos[MAX_COL];
4    T[0].row = nc;  T[0].col = M[0].row;
5    T[0].value = nt;
6    if (nt <= 0) return;                    /* empty */
7    for (i = 0; i < nc; i++) row_terms[i] = 0;
8    for (i = 1; i <= nt; i++) row_terms[M[i].col]++;
9    start_pos[0] = 1;
▶    for (i = 1; i < nc; i++)
▶      start_pos[i] = start_pos[i-1] + row_terms[i-1];
12   /* phase 2: put the terms in place */
```

VARIABLES

nc = 4 nt = 4 i = 1

fast_transpose (1): Count and Find the Starts

	row	col	value
M[0]	3	4	4
M[1]	0	0	7
M[2]	0	3	3
M[3]	1	1	4
M[4]	2	0	2

	column 0	column 1	column 2	column 3
row_terms	2	1	0	1
start_pos	1	3	4	

start_pos[2] = 3 + 1 = **4**.

Step 9 / 10

```
1  void fast_transpose(term M[], term T[]) {
2    int i, j, nc = M[0].col, nt = M[0].value;
3    int row_terms[MAX_COL], start_pos[MAX_COL];
4    T[0].row = nc;  T[0].col = M[0].row;
5    T[0].value = nt;
6    if (nt <= 0) return;                    /* empty */
7    for (i = 0; i < nc; i++) row_terms[i] = 0;
8    for (i = 1; i <= nt; i++) row_terms[M[i].col]++;
9    start_pos[0] = 1;
▶    for (i = 1; i < nc; i++)
▶      start_pos[i] = start_pos[i-1] + row_terms[i-1];
12   /* phase 2: put the terms in place */
```

VARIABLES

nc = 4 | nt = 4 | i = 2

fast_transpose (1): Count and Find the Starts

	row	col	value
M[0]	3	4	4
M[1]	0	0	7
M[2]	0	3	3
M[3]	1	1	4
M[4]	2	0	2

	column 0	column 1	column 2	column 3
row_terms	2	1	0	1
start_pos	1	3	4	4

start_pos[3] = 4 + 0 = **4** (column 2 is empty, so column 3 also starts at 4).

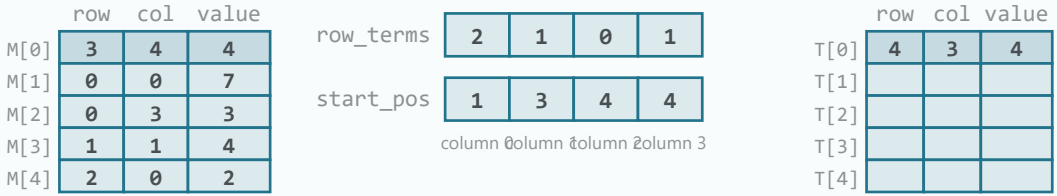
Step 10 / 10

```
1  void fast_transpose(term M[], term T[]) {
2    int i, j, nc = M[0].col, nt = M[0].value;
3    int row_terms[MAX_COL], start_pos[MAX_COL];
4    T[0].row = nc;  T[0].col = M[0].row;
5    T[0].value = nt;
6    if (nt <= 0) return;                    /* empty */
7    for (i = 0; i < nc; i++) row_terms[i] = 0;
8    for (i = 1; i <= nt; i++) row_terms[M[i].col]++;
9    start_pos[0] = 1;
▶    for (i = 1; i < nc; i++)
▶      start_pos[i] = start_pos[i-1] + row_terms[i-1];
12   /* phase 2: put the terms in place */
```

VARIABLES

nc = 4 nt = 4 i = 3

fast_transpose (2): Put the Terms in Place



M is scanned once, start to end; each term goes straight to its place.

Step 1 / 10

```
1   /* phase 1: row_terms and start_pos ready */
▶   for (i = 1; i <= nt; i++) {
3     j = start_pos[M[i].col]++;  /* read, bump */
4     T[j].row = M[i].col;
5     T[j].col = M[i].row;
6     T[j].value = M[i].value;
7   }
8 }
```

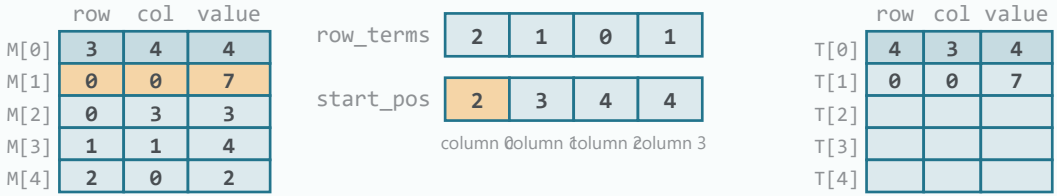
VARIABLES

`nt = 4`

`start_pos = [1, 3, 4, 4]`

Instructor *"A programmer must know the operator precedence of their language well."*

fast_transpose (2): Put the Terms in Place



M[1] = (0, 0, 7): **j = start_pos[0]++** → j = 1, start_pos[0] = 2.

Step 2 / 10

```
1  /* phase 1: row_terms and start_pos ready */
2  for (i = 1; i <= nt; i++) {
▶    j = start_pos[M[i].col]++;  /* read, bump */
4    T[j].row = M[i].col;
5    T[j].col = M[i].row;
6    T[j].value = M[i].value;
7  }
8 }
```

VARIABLES

`nt = 4` `i = 1` `M[i].col = 0` `j = 1` `start_pos = [2, 3, 4, 4]`

Instructor *"A programmer must know the operator precedence of their language well."*

fast_transpose (2): Put the Terms in Place

	row	col	value
M[0]	3	4	4
M[1]	0	0	7
M[2]	0	3	3
M[3]	1	1	4
M[4]	2	0	2

	column 0	column 1	column 2	column 3
row_terms	2	1	0	1
start_pos	2	3	4	4

	row	col	value
T[0]	4	3	4
T[1]	0	0	7
T[2]			
T[3]			
T[4]			

T[1] = (0, 0, 7): row and column swapped.

Step 3 / 10

```
1   /* phase 1: row_terms and start_pos ready */
2   for (i = 1; i <= nt; i++) {
3     j = start_pos[M[i].col]++;  /* read, bump */
▶     T[j].row = M[i].col;
▶     T[j].col = M[i].row;
▶     T[j].value = M[i].value;
7   }
8 }
```

VARIABLES

`nt = 4` `i = 1` `M[i].col = 0` `j = 1` `start_pos = [2, 3, 4, 4]`

Instructor *"A programmer must know the operator precedence of their language well."*

fast_transpose (2): Put the Terms in Place

	row	col	value
M[0]	3	4	4
M[1]	0	0	7
M[2]	0	3	3
M[3]	1	1	4
M[4]	2	0	2

	column 0	column 1	column 2	column 3
row_terms	2	1	0	1
start_pos	2	3	4	5

	row	col	value
T[0]	4	3	4
T[1]	0	0	7
T[2]			
T[3]			
T[4]	3	0	3

M[2] = (0, 3, 3): **j = start_pos[3]++** → j = 4, start_pos[3] = 5.

Step 4 / 10

```
1  /* phase 1: row_terms and start_pos ready */
2  for (i = 1; i <= nt; i++) {
▶    j = start_pos[M[i].col]++;  /* read, bump */
4    T[j].row = M[i].col;
5    T[j].col = M[i].row;
6    T[j].value = M[i].value;
7  }
8 }
```

VARIABLES

`nt = 4` `i = 2` `M[i].col = 3` `j = 4` `start_pos = [2, 3, 4, 5]`

Instructor *"A programmer must know the operator precedence of their language well."*

fast_transpose (2): Put the Terms in Place

	row	col	value
M[0]	3	4	4
M[1]	0	0	7
M[2]	0	3	3
M[3]	1	1	4
M[4]	2	0	2

	column 0	column 1	column 2	column 3
row_terms	2	1	0	1
start_pos	2	3	4	5

	row	col	value
T[0]	4	3	4
T[1]	0	0	7
T[2]			
T[3]			
T[4]	3	0	3

T[4] = (3, 0, 3): row and column swapped.

Step 5 / 10

```
1   /* phase 1: row_terms and start_pos ready */
2   for (i = 1; i <= nt; i++) {
3     j = start_pos[M[i].col]++;  /* read, bump */
      T[j].row = M[i].col;
      T[j].col = M[i].row;
      T[j].value = M[i].value;
7   }
8 }
```

VARIABLES

`nt = 4` `i = 2` `M[i].col = 3` `j = 4` `start_pos = [2, 3, 4, 5]`

Instructor *"A programmer must know the operator precedence of their language well."*

fast_transpose (2): Put the Terms in Place

	row	col	value
M[0]	3	4	4
M[1]	0	0	7
M[2]	0	3	3
M[3]	1	1	4
M[4]	2	0	2

	column 0	column 1	column 2	column 3
row_terms	2	1	0	1
start_pos	2	4	4	5

	row	col	value
T[0]	4	3	4
T[1]	0	0	7
T[2]			
T[3]	1	1	4
T[4]	3	0	3

M[3] = (1, 1, 4): **j = start_pos[1]++** → j = 3, start_pos[1] = 4.

Step 6 / 10

```
1  /* phase 1: row_terms and start_pos ready */
2  for (i = 1; i <= nt; i++) {
▶    j = start_pos[M[i].col]++;  /* read, bump */
4    T[j].row = M[i].col;
5    T[j].col = M[i].row;
6    T[j].value = M[i].value;
7  }
8 }
```

VARIABLES

`nt = 4` `i = 3` `M[i].col = 1` `j = 3` `start_pos = [2, 4, 4, 5]`

Instructor *"A programmer must know the operator precedence of their language well."*

fast_transpose (2): Put the Terms in Place

	row	col	value
M[0]	3	4	4
M[1]	0	0	7
M[2]	0	3	3
M[3]	1	1	4
M[4]	2	0	2

	column 0	column 1	column 2	column 3
row_terms	2	1	0	1
start_pos	2	4	4	5

	row	col	value
T[0]	4	3	4
T[1]	0	0	7
T[2]			
T[3]	1	1	4
T[4]	3	0	3

T[3] = (1, 1, 4): row and column swapped.

Step 7 / 10

```
1   /* phase 1: row_terms and start_pos ready */
2   for (i = 1; i <= nt; i++) {
3     j = start_pos[M[i].col]++;  /* read, bump */
      T[j].row = M[i].col;
      T[j].col = M[i].row;
      T[j].value = M[i].value;
7   }
8 }
```

VARIABLES

`nt = 4` `i = 3` `M[i].col = 1` `j = 3` `start_pos = [2, 4, 4, 5]`

Instructor *"A programmer must know the operator precedence of their language well."*

fast_transpose (2): Put the Terms in Place

	row	col	value
M[0]	3	4	4
M[1]	0	0	7
M[2]	0	3	3
M[3]	1	1	4
M[4]	2	0	2

	column 0	column 1	column 2	column 3
row_terms	2	1	0	1
start_pos	3	4	4	5

	row	col	value
T[0]	4	3	4
T[1]	0	0	7
T[2]	0	2	2
T[3]	1	1	4
T[4]	3	0	3

M[4] = (2, 0, 2): **j = start_pos[0]++** → j = 2, start_pos[0] = 3.

Step 8 / 10

```
1  /* phase 1: row_terms and start_pos ready */
2  for (i = 1; i <= nt; i++) {
▶    j = start_pos[M[i].col]++;  /* read, bump */
4    T[j].row = M[i].col;
5    T[j].col = M[i].row;
6    T[j].value = M[i].value;
7  }
8 }
```

VARIABLES

`nt = 4` `i = 4` `M[i].col = 0` `j = 2` `start_pos = [3, 4, 4, 5]`

Instructor *"A programmer must know the operator precedence of their language well."*

fast_transpose (2): Put the Terms in Place

	row	col	value
M[0]	3	4	4
M[1]	0	0	7
M[2]	0	3	3
M[3]	1	1	4
M[4]	2	0	2

	column 0	column 1	column 2	column 3
row_terms	2	1	0	1
start_pos	3	4	4	5

	row	col	value
T[0]	4	3	4
T[1]	0	0	7
T[2]	0	2	2
T[3]	1	1	4
T[4]	3	0	3

T[2] = (0, 2, 2): row and column swapped.

Step 9 / 10

```
1  /* phase 1: row_terms and start_pos ready */
2  for (i = 1; i <= nt; i++) {
3    j = start_pos[M[i].col]++;  /* read, bump */
▶    T[j].row = M[i].col;
▶    T[j].col = M[i].row;
▶    T[j].value = M[i].value;
7  }
8 }
```

VARIABLES

`nt = 4` `i = 4` `M[i].col = 0` `j = 2` `start_pos = [3, 4, 4, 5]`

Instructor *"A programmer must know the operator precedence of their language well."*

fast_transpose (2): Put the Terms in Place

	row	col	value
M[0]	3	4	4
M[1]	0	0	7
M[2]	0	3	3
M[3]	1	1	4
M[4]	2	0	2

	column 0	column 1	column 2	column 3
row_terms	2	1	0	1
start_pos	3	4	4	5

	row	col	value
T[0]	4	3	4
T[1]	0	0	7
T[2]	0	2	2
T[3]	1	1	4
T[4]	3	0	3

Done — the same result as transpose. Two linear scans: **O(cols + terms)**.

Step 10 / 10

```
1   /* phase 1: row_terms and start_pos ready */
2   for (i = 1; i <= nt; i++) {
3     j = start_pos[M[i].col]++;  /* read, bump */
4     T[j].row = M[i].col;
5     T[j].col = M[i].row;
6     T[j].value = M[i].value;
7   }
▶ }
```

VARIABLES

`nt = 4`

`start_pos = [3, 4, 4, 5]`

Instructor *"A programmer must know the operator precedence of their language well."*

Analysis, and a Version with One Array

Loops: num_cols + num_terms + num_cols + num_terms (+ constants):

2·terms + 2·cols + 3 = O(cols + terms)

For a dense matrix O(rows·cols): the same order as transposing a 2D array. The price: two auxiliary arrays (O(cols) space).

> **Instructor** *"Remember I said a fast transpose can be done with a single array? Here it is..."*

```
/* turn row_terms into start_pos in place */
for (i = 0, pos = 1; i < nc; i++) {
  int c = row_terms[i];  /* terms in this column */
  row_terms[i] = pos;  /* now a start position */
  pos += c;
}
for (i = 1; i <= nt; i++) {
  j = row_terms[M[i].col]++;
  T[j].row = M[i].col;  T[j].col = M[i].row;
  T[j].value = M[i].value;
}
```

The fast transpose can also be written with a single array. It is exactly **counting sort**, with the column number as the key.

$D_{m\times p} = A_{m\times n} \cdot B_{n\times p}$

$$d_{ij} = \Sigma_{k=0}^{n-1} a_{ik} \cdot b_{kj}$$

Row i of A and column j of B are "multiplied and summed". The product of two sparse matrices **need not be sparse**:

A

1	0	0
1	0	0
1	0	0

·

B

1	1	1
0	0	0
0	0	0

=

D

1	1	1
1	1	1
1	1	1

- In the triples, row i of A is **contiguous**, but column j of B is scattered.
- So we first take the **transpose of B** (fast_transpose): B's columns become consecutive rows of B^T.
- Then each row of A is multiplied by each row of B^T by merging on the **col** field.

> Saying "first transpose the **first** matrix" is wrong; in Horowitz's `mmult` (and logically) it is the **second matrix B** that is transposed.

Helpers: End of Row, dot Product, store

```c
/* where the row of the term at s ends */
int row_end(term t[], int s) {
  int e = s;
  while (e <= t[0].value &&
         t[e].row == t[s].row) e++;
  return e;
}
/* row a[x..ex) · row b[y..ey) */
int dot(term a[], int x, int ex,
        term b[], int y, int ey) {
  int sum = 0;
  while (x < ex && y < ey)
    if (a[x].col < b[y].col) x++;
    else if (a[x].col > b[y].col) y++;
    else sum += a[x++].value * b[y++].value;
  return sum;
}
```

```c
void store(term d[], int *nd,
      int r, int c, int s) {
  if (s == 0) return;
  if (++*nd >= MAX_TERMS) {
    fprintf(stderr, "Too many terms\n");
    exit(1);
  }
  d[*nd].row = r;
  d[*nd].col = c;
  d[*nd].value = s;
}
```

`dot` is a two-pointer merge like padd: only terms with the same **k** (col) are multiplied. A zero sum is not stored.

Sparse Multiplication: Row of A · Row of Bᵀ

A (2×3)

	row	col	value
a[0]	2	3	3
a[1]	0	0	1
a[2]	0	2	2
a[3]	1	1	3

Bᵀ (B: 3×2)

	row	col	value
bt[0]	2	3	3
bt[1]	0	0	4
bt[2]	0	2	6
bt[3]	1	1	5

D = A·B (2×2)

	row	col	value
d[0]	2	2	0
d[1]			
d[2]			

A = [[1,0,2],[0,3,0]], B = [[4,0],[0,5],[6,0]]; the transpose of B, `bt`, was computed beforehand.

Step 1 / 12

```
1  void smult(term a[], term bt[], term d[]) {
▶    int ra, ea, rb, eb, nd = 0;
3    for (ra = 1; ra <= a[0].value; ra = ea) {
4      ea = row_end(a, ra);        /* row of A [ra,ea) */
5      for (rb = 1; rb <= bt[0].value; rb = eb) {
6        eb = row_end(bt, rb);     /* col of B [rb,eb) */
7        store(d, &nd, a[ra].row, bt[rb].row,
8              dot(a, ra, ea, bt, rb, eb));
9      }
10   }
11   d[0].row = a[0].row;  d[0].col = bt[0].row;
12   d[0].value = nd;
13 }
```

VARIABLES

nd = 0

Sparse Multiplication: Row of A · Row of Bᵀ

A (2×3)

	row	col	value
a[0]	2	3	3
a[1]	0	0	1
a[2]	0	2	2
a[3]	1	1	3

Bᵀ (B: 3×2)

	row	col	value
bt[0]	2	3	3
bt[1]	0	0	4
bt[2]	0	2	6
bt[3]	1	1	5

D = A·B (2×2)

	row	col	value
d[0]	2	2	0
d[1]			
d[2]			

Row 0 of A: a[1..2].

Step 2 / 12

```
1  void smult(term a[], term bt[], term d[]) {
2    int ra, ea, rb, eb, nd = 0;
3    for (ra = 1; ra <= a[0].value; ra = ea) {
▶      ea = row_end(a, ra);        /* row of A [ra,ea) */
5      for (rb = 1; rb <= bt[0].value; rb = eb) {
6        eb = row_end(bt, rb);     /* col of B [rb,eb) */
7        store(d, &nd, a[ra].row, bt[rb].row,
8              dot(a, ra, ea, bt, rb, eb));
9      }
10   }
11   d[0].row = a[0].row;  d[0].col = bt[0].row;
12   d[0].value = nd;
13 }
```

VARIABLES

ra = 1 ea = 3 nd = 0

Sparse Multiplication: Row of A · Row of Bᵀ

A (2×3)

	row	col	value
a[0]	2	3	3
a[1]	0	0	1
a[2]	0	2	2
a[3]	1	1	3

Bᵀ (B: 3×2)

	row	col	value
bt[0]	2	3	3
bt[1]	0	0	4
bt[2]	0	2	6
bt[3]	1	1	5

D = A·B (2×2)

	row	col	value
d[0]	2	2	0
d[1]			
d[2]			

Column 0 of B = bt[1..2].

Step 3 / 12

```
1  void smult(term a[], term bt[], term d[]) {
2    int ra, ea, rb, eb, nd = 0;
3    for (ra = 1; ra <= a[0].value; ra = ea) {
4      ea = row_end(a, ra);        /* row of A [ra,ea) */
5      for (rb = 1; rb <= bt[0].value; rb = eb) {
▶        eb = row_end(bt, rb);     /* col of B [rb,eb) */
7        store(d, &nd, a[ra].row, bt[rb].row,
8              dot(a, ra, ea, bt, rb, eb));
9      }
10   }
11   d[0].row = a[0].row;  d[0].col = bt[0].row;
12   d[0].value = nd;
13 }
```

VARIABLES

ra = 1 · ea = 3 · rb = 1 · eb = 3 · nd = 0

Sparse Multiplication: Row of A · Row of Bᵀ

A (2×3)

	row	col	value
a[0]	2	3	3
a[1]	0	0	1
a[2]	0	2	2
a[3]	1	1	3

Bᵀ (B: 3×2)

	row	col	value
bt[0]	2	3	3
bt[1]	0	0	4
bt[2]	0	2	6
bt[3]	1	1	5

D = A·B (2×2)

	row	col	value
d[0]	2	2	1
d[1]	0	0	16
d[2]			

dot: 1·4 + 2·6 = **16** → d[1] = (0, 0, 16).

Step 4 / 12

```
1  void smult(term a[], term bt[], term d[]) {
2    int ra, ea, rb, eb, nd = 0;
3    for (ra = 1; ra <= a[0].value; ra = ea) {
4      ea = row_end(a, ra);        /* row of A [ra,ea) */
5      for (rb = 1; rb <= bt[0].value; rb = eb) {
6        eb = row_end(bt, rb);     /* col of B [rb,eb) */
▶        store(d, &nd, a[ra].row, bt[rb].row,
▶              dot(a, ra, ea, bt, rb, eb));
9      }
10   }
11   d[0].row = a[0].row;  d[0].col = bt[0].row;
12   d[0].value = nd;
13 }
```

VARIABLES

ra = 1 · ea = 3 · rb = 1 · eb = 3 · nd = 1

Sparse Multiplication: Row of A · Row of Bᵀ

A (2×3)

	row	col	value
a[0]	2	3	3
a[1]	0	0	1
a[2]	0	2	2
a[3]	1	1	3

Bᵀ (B: 3×2)

	row	col	value
bt[0]	2	3	3
bt[1]	0	0	4
bt[2]	0	2	6
bt[3]	1	1	5

D = A·B (2×2)

	row	col	value
d[0]	2	2	1
d[1]	0	0	16
d[2]			

Column 1 of B = bt[3..3].

Step 5 / 12

```
1  void smult(term a[], term bt[], term d[]) {
2    int ra, ea, rb, eb, nd = 0;
3    for (ra = 1; ra <= a[0].value; ra = ea) {
4      ea = row_end(a, ra);        /* row of A [ra,ea) */
5      for (rb = 1; rb <= bt[0].value; rb = eb) {
▶        eb = row_end(bt, rb);     /* col of B [rb,eb) */
7        store(d, &nd, a[ra].row, bt[rb].row,
8              dot(a, ra, ea, bt, rb, eb));
9      }
10   }
11   d[0].row = a[0].row;  d[0].col = bt[0].row;
12   d[0].value = nd;
13 }
```

VARIABLES

ra = 1 · ea = 3 · rb = 3 · eb = 4 · nd = 1

Sparse Multiplication: Row of A · Row of Bᵀ

A (2×3)

	row	col	value
a[0]	2	3	3
a[1]	0	0	1
a[2]	0	2	2
a[3]	1	1	3

Bᵀ (B: 3×2)

	row	col	value
bt[0]	2	3	3
bt[1]	0	0	4
bt[2]	0	2	6
bt[3]	1	1	5

D = A·B (2×2)

	row	col	value
d[0]	2	2	1
d[1]	0	0	16
d[2]			

dot: no common col = **0** → zero, **not stored**.

Step 6 / 12

```
1  void smult(term a[], term bt[], term d[]) {
2    int ra, ea, rb, eb, nd = 0;
3    for (ra = 1; ra <= a[0].value; ra = ea) {
4      ea = row_end(a, ra);        /* row of A [ra,ea) */
5      for (rb = 1; rb <= bt[0].value; rb = eb) {
6        eb = row_end(bt, rb);     /* col of B [rb,eb) */
▶        store(d, &nd, a[ra].row, bt[rb].row,
▶              dot(a, ra, ea, bt, rb, eb));
9      }
10   }
11   d[0].row = a[0].row;  d[0].col = bt[0].row;
12   d[0].value = nd;
13 }
```

VARIABLES

ra = 1 · ea = 3 · rb = 3 · eb = 4 · nd = 1

Sparse Multiplication: Row of A · Row of Bᵀ

A (2×3)

	row	col	value
a[0]	2	3	3
a[1]	0	0	1
a[2]	0	2	2
a[3]	1	1	3

Bᵀ (B: 3×2)

	row	col	value
bt[0]	2	3	3
bt[1]	0	0	4
bt[2]	0	2	6
bt[3]	1	1	5

D = A·B (2×2)

	row	col	value
d[0]	2	2	1
d[1]	0	0	16
d[2]			

Row 1 of A: a[3..3].

Step 7 / 12

```
1  void smult(term a[], term bt[], term d[]) {
2    int ra, ea, rb, eb, nd = 0;
3    for (ra = 1; ra <= a[0].value; ra = ea) {
▶      ea = row_end(a, ra);        /* row of A [ra,ea) */
5      for (rb = 1; rb <= bt[0].value; rb = eb) {
6        eb = row_end(bt, rb);     /* col of B [rb,eb) */
7        store(d, &nd, a[ra].row, bt[rb].row,
8              dot(a, ra, ea, bt, rb, eb));
9      }
10   }
11   d[0].row = a[0].row;  d[0].col = bt[0].row;
12   d[0].value = nd;
13 }
```

VARIABLES

ra = 3 ea = 4 nd = 1

Sparse Multiplication: Row of A · Row of Bᵀ

A (2×3)

	row	col	value
a[0]	2	3	3
a[1]	0	0	1
a[2]	0	2	2
a[3]	1	1	3

Bᵀ (B: 3×2)

	row	col	value
bt[0]	2	3	3
bt[1]	0	0	4
bt[2]	0	2	6
bt[3]	1	1	5

D = A·B (2×2)

	row	col	value
d[0]	2	2	1
d[1]	0	0	16
d[2]			

Column 0 of B = bt[1..2].

Step 8 / 12

```
1  void smult(term a[], term bt[], term d[]) {
2    int ra, ea, rb, eb, nd = 0;
3    for (ra = 1; ra <= a[0].value; ra = ea) {
4      ea = row_end(a, ra);        /* row of A [ra,ea) */
5      for (rb = 1; rb <= bt[0].value; rb = eb) {
▶        eb = row_end(bt, rb);     /* col of B [rb,eb) */
7        store(d, &nd, a[ra].row, bt[rb].row,
8              dot(a, ra, ea, bt, rb, eb));
9      }
10   }
11   d[0].row = a[0].row;  d[0].col = bt[0].row;
12   d[0].value = nd;
13 }
```

VARIABLES

ra = 3 · ea = 4 · rb = 1 · eb = 3 · nd = 1

Sparse Multiplication: Row of A · Row of Bᵀ

A (2×3)

	row	col	value
a[0]	2	3	3
a[1]	0	0	1
a[2]	0	2	2
a[3]	1	1	3

Bᵀ (B: 3×2)

	row	col	value
bt[0]	2	3	3
bt[1]	0	0	4
bt[2]	0	2	6
bt[3]	1	1	5

D = A·B (2×2)

	row	col	value
d[0]	2	2	1
d[1]	0	0	16
d[2]			

dot: no common col = **0** → zero, **not stored**.

Step 9 / 12

```
1  void smult(term a[], term bt[], term d[]) {
2    int ra, ea, rb, eb, nd = 0;
3    for (ra = 1; ra <= a[0].value; ra = ea) {
4      ea = row_end(a, ra);        /* row of A [ra,ea) */
5      for (rb = 1; rb <= bt[0].value; rb = eb) {
6        eb = row_end(bt, rb);     /* col of B [rb,eb) */
▶        store(d, &nd, a[ra].row, bt[rb].row,
▶              dot(a, ra, ea, bt, rb, eb));
9      }
10   }
11   d[0].row = a[0].row;  d[0].col = bt[0].row;
12   d[0].value = nd;
13 }
```

VARIABLES

ra = 3 ea = 4 rb = 1 eb = 3 nd = 1

Sparse Multiplication: Row of A · Row of Bᵀ

A (2×3)

	row	col	value
a[0]	2	3	3
a[1]	0	0	1
a[2]	0	2	2
a[3]	1	1	3

Bᵀ (B: 3×2)

	row	col	value
bt[0]	2	3	3
bt[1]	0	0	4
bt[2]	0	2	6
bt[3]	1	1	5

D = A·B (2×2)

	row	col	value
d[0]	2	2	1
d[1]	0	0	16
d[2]			

Column 1 of B = bt[3..3].

Step 10 / 12

```
1  void smult(term a[], term bt[], term d[]) {
2    int ra, ea, rb, eb, nd = 0;
3    for (ra = 1; ra <= a[0].value; ra = ea) {
4      ea = row_end(a, ra);        /* row of A [ra,ea) */
5      for (rb = 1; rb <= bt[0].value; rb = eb) {
▶        eb = row_end(bt, rb);     /* col of B [rb,eb) */
7        store(d, &nd, a[ra].row, bt[rb].row,
8              dot(a, ra, ea, bt, rb, eb));
9      }
10   }
11   d[0].row = a[0].row;  d[0].col = bt[0].row;
12   d[0].value = nd;
13 }
```

VARIABLES

ra = 3 · ea = 4 · rb = 3 · eb = 4 · nd = 1

Sparse Multiplication: Row of A · Row of Bᵀ

A (2×3)

	row	col	value
a[0]	2	3	3
a[1]	0	0	1
a[2]	0	2	2
a[3]	1	1	3

Bᵀ (B: 3×2)

	row	col	value
bt[0]	2	3	3
bt[1]	0	0	4
bt[2]	0	2	6
bt[3]	1	1	5

D = A·B (2×2)

	row	col	value
d[0]	2	2	2
d[1]	0	0	16
d[2]	1	1	15

dot: 3·5 = **15** → d[2] = (1, 1, 15).

Step 11 / 12

```
1  void smult(term a[], term bt[], term d[]) {
2    int ra, ea, rb, eb, nd = 0;
3    for (ra = 1; ra <= a[0].value; ra = ea) {
4      ea = row_end(a, ra);        /* row of A [ra,ea) */
5      for (rb = 1; rb <= bt[0].value; rb = eb) {
6        eb = row_end(bt, rb);     /* col of B [rb,eb) */
▶        store(d, &nd, a[ra].row, bt[rb].row,
▶              dot(a, ra, ea, bt, rb, eb));
9      }
10   }
11   d[0].row = a[0].row;  d[0].col = bt[0].row;
12   d[0].value = nd;
13 }
```

VARIABLES

ra = 3 · ea = 4 · rb = 3 · eb = 4 · nd = 2

Sparse Multiplication: Row of A · Row of Bᵀ

A (2×3)

	row	col	value
a[0]	2	3	3
a[1]	0	0	1
a[2]	0	2	2
a[3]	1	1	3

Bᵀ (B: 3×2)

	row	col	value
bt[0]	2	3	3
bt[1]	0	0	4
bt[2]	0	2	6
bt[3]	1	1	5

D = A·B (2×2)

	row	col	value
d[0]	2	2	2
d[1]	0	0	16
d[2]	1	1	15

Write the header: D is 2×2 with 2 terms. D = [[16, 0], [0, 15]].

Step 12 / 12

```
1  void smult(term a[], term bt[], term d[]) {
2    int ra, ea, rb, eb, nd = 0;
3    for (ra = 1; ra <= a[0].value; ra = ea) {
4      ea = row_end(a, ra);        /* row of A [ra,ea) */
5      for (rb = 1; rb <= bt[0].value; rb = eb) {
6        eb = row_end(bt, rb);     /* col of B [rb,eb) */
7        store(d, &nd, a[ra].row, bt[rb].row,
8              dot(a, ra, ea, bt, rb, eb));
9      }
10   }
▶    d[0].row = a[0].row;  d[0].col = bt[0].row;
▶    d[0].value = nd;
13 }
```

VARIABLES

nd = 2

The Complexity of Sparse Multiplication

- Transpose: $O(cols_B + t_B)$.
- For each row i of A, B^T is walked end to end: $\Sigma_i\ t_B = \mathbf{rows_A \cdot t_B}$.
- For each pair (i, j) the terms of A's row are walked too: $\Sigma_j\ t_A = \mathbf{cols_B \cdot t_A}$.
- Total: $\mathbf{O(cols_B \cdot t_A + rows_A \cdot t_B)}$.

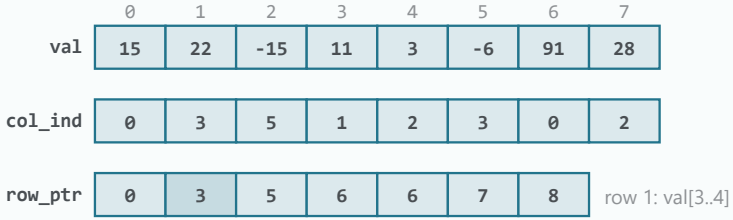
Classic multiplication

$O(rows_A \cdot cols_A \cdot cols_B)$. For dense matrices ($t_A = rows_A \cdot cols_A$) the sparse version reaches the same order with a larger constant.

When is it worth it?

When $t_A \ll rows \cdot cols$ and $t_B \ll rows \cdot cols$. The result D is not guaranteed to be sparse: MAX_TERMS may be exceeded.

Compressed Sparse Row (CSR)



The same 6×6 matrix: 8 + 8 + 7 = 23 integers (27 as triples). Row i: `val[row_ptr[i] .. row_ptr[i+1]-1]`; the empty row 3 → two equal consecutive row_ptr values.

```
/* y = M * x  (matrix-vector product) */
for (i = 0; i < rows; i++) {
  y[i] = 0;
  for (k = row_ptr[i]; k < row_ptr[i+1]; k++)
    y[i] += val[k] * x[col_ind[k]];
}
```

Use

One of the most common formats in scientific computing; M·x is O(rows + terms). The column version: CSC.

Connection

row_ptr is the same idea as start_pos in fast_transpose (prefix sums).

Representing a Sparse Matrix with Linked Lists

- In a triple array, insertion/deletion needs shifting and there is a MAX_TERMS limit.
- In the linked representation each nonzero entry is a node; entries in the same row and in the same column are joined by **circular lists**, with row/column header nodes (Horowitz 4.7).
- Insert/delete is an O(1) link change (once the place is found); traversal by row and by column is easy.
- The price: two extra pointers per entry.

Instructor *"If I hold your homework over mine against the light… (there should be no difference)"*

Where?

The linked implementation of sparse matrices is a direct application of linked lists.

Complexity Summary

Operation	Time	Note
padd (n, m terms)	O(n + m)	sorted terms + merging
pmult (array, repeated padd)	$O(n^2m)$	smaller one outside: $O(\min^2 \cdot \max)$
poly_mult (linked list)	$O(n^2m)$ worst case	start shortens the search
transpose	O(cols·terms)	$O(cols^2 \cdot rows)$ for a dense matrix
fast_transpose	O(cols + terms)	O(cols) extra space
Sparse multiplication (smult / mmult)	$O(cols_B \cdot t_A + rows_A \cdot t_B)$	B^T first
M·x with CSR	O(rows + terms)	row start array

WRAP-UP

Common Mistakes

1. Keeping the terms unsorted but still writing padd with two pointers.
2. Appending a term whose sum is 0, or not deleting it from the list.
3. Truncating a float coefficient through an int parameter in `attach`; not checking `avail` for overflow.
4. Only swapping row/col in a transpose and forgetting the order; mistaking the header M[0] for a term.
5. Mixing up prefix and postfix in `start_pos[M[i].col]++`.
6. Transposing A in sparse multiplication; assuming the result stays sparse.

Exercises

1. Trace padd for $A = 2x^4 + 3x^2 - 1$ and $B = -3x^2 + x + 1$. How many terms does D have?
2. In the P array representation, at most how many attach calls does adding two 10-term polynomials make?
3. In a 5×5 matrix, from how many nonzero entries on does the triple representation take more space than a 2D array?
4. What are row_terms and start_pos for the 3×4 matrix with terms (0,1,5), (1,0,2), (1,3,4), (2,1,1)?
5. How is fast_transpose written with a single array? Which sorting algorithm does it resemble?
6. Which term does poly_mult delete when computing $(x^2 + 1)(x^2 - 1)$?

▶ **Hints / answers**

WRAP-UP

Chapter Summary

Representation matters

A coefficient array wastes space on a sparse polynomial; an array of terms keeps only the meaningful ones.

Order + merging

padd is O(n + m): walk two sorted arrays with two pointers.

Linked lists

Flexible when the number of terms is unknown; poly_mult combines equal exponents and deletes zeros.

Triples

⟨row, col, value⟩ + header M[0]; pays off while t < rows·cols/3.

Transpose

transpose is O(cols·t); fast_transpose counts and places in O(cols + t).

Multiplication

B^T first, then row-by-row merging; CSR is the standard form of the same “row start” idea.

References and Next Steps

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., 2.4–2.5 (polynomials, sparse matrices), 4.4 and 4.7 (linked polynomials and sparse matrices).
- Y. Saad, *Iterative Methods for Sparse Linear Systems*, 3.4 (CSR/CSC formats).

← Chapter 4: Linked Lists

Course page

Chapter 6: Stacks

The End

Chapter 5: Polynomials and Sparse Matrices

Next chapter: Chapter 6: Stacks · Course page