```
5       for (i = 0; i < dim; i++)
6           for (j = 0; j < dim; j++)
7               dst[j*dim + i] = src[i*dim + j];
8   }
```

where the arguments to the procedure are pointers to the destination (`dst`) and source (`src`) matrices, as well as the matrix size N (`dim`). Your job is to devise a transpose routine that runs as fast as possible.

6.46 ◆◆◆◆

This assignment is an intriguing variation of Problem 6.45. Consider the problem of converting a directed graph g into its undirected counterpart g'. The graph g' has an edge from vertex u to vertex v if and only if there is an edge from u to v or from v to u in the original graph g. The graph g is represented by its *adjacency matrix* G as follows. If N is the number of vertices in g, then G is an $N \times N$ matrix and its entries are all either 0 or 1. Suppose the vertices of g are named $v_0, v_1, v_2, \ldots, v_{N-1}$. Then $G[i][j]$ is 1 if there is an edge from v_i to v_j and is 0 otherwise. Observe that the elements on the diagonal of an adjacency matrix are always 1 and that the adjacency matrix of an undirected graph is symmetric. This code can be written with a simple loop:

```
1   void col_convert(int *G, int dim) {
2       int i, j;
3
4       for (i = 0; i < dim; i++)
5           for (j = 0; j < dim; j++)
6               G[j*dim + i] = G[j*dim + i] || G[i*dim + j];
7   }
```

Your job is to devise a conversion routine that runs as fast as possible. As before, you will need to apply concepts you learned in Chapters 5 and 6 to come up with a good solution.

Solutions to Practice Problems

Solution to Problem 6.1 (page 620)

The idea here is to minimize the number of address bits by minimizing the aspect ratio $\max(r, c)/\min(r, c)$. In other words, the squarer the array, the fewer the address bits.

Organization	r	c	b_r	b_c	$\max(b_r, b_c)$
16×1	4	4	2	2	2
16×4	4	4	2	2	2
128×8	16	8	4	3	4
512×4	32	16	5	4	5
$1{,}024 \times 4$	32	32	5	5	5

Solution to Problem 6.2 (page 628)

The point of this little drill is to make sure you understand the relationship between cylinders and tracks. Once you have that straight, just plug and chug:

$$\begin{aligned}\text{Disk capacity} &= \frac{1{,}024 \text{ bytes}}{\text{sector}} \times \frac{500 \text{ sectors}}{\text{track}} \times \frac{15{,}000 \text{ tracks}}{\text{surface}} \times \frac{2 \text{ surfaces}}{\text{platter}} \times \frac{3 \text{ platters}}{\text{disk}} \\ &= 46{,}080{,}000{,}000 \text{ bytes} \\ &= 46.08 \text{ GB}\end{aligned}$$

Solution to Problem 6.3 (page 631)

The solution to this problem is a straightforward application of the formula for disk access time. The average rotational latency (in ms) is

$$\begin{aligned}T_{\text{avg rotation}} &= 1/2 \times T_{\text{max rotation}} \\ &= 1/2 \times (60 \text{ secs}/12{,}000 \text{ RPM}) \times 1{,}000 \text{ ms/sec} \\ &\approx 2.5 \text{ ms}\end{aligned}$$

The average transfer time is

$$\begin{aligned}T_{\text{avg transfer}} &= (60 \text{ secs}/12{,}000 \text{ RPM}) \times 1/300 \text{ sectors/track} \times 1{,}000 \text{ ms/sec} \\ &\approx 0.016 \text{ ms}\end{aligned}$$

Putting it all together, the total estimated access time is

$$\begin{aligned}T_{\text{access}} &= T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}} \\ &= 5 \text{ ms} + 2.5 \text{ ms} + 0.016 \text{ ms} \\ &\approx 7.516 \text{ ms}\end{aligned}$$

Solution to Problem 6.4 (page 631)

This is a good check of your understanding of the factors that affect disk performance. First we need to determine a few basic properties of the file and the disk. The file consists of 10,000 512-byte logical blocks. For the disk, $T_{\text{avg seek}} = 6$ ms, $T_{\text{max rotation}} = 4.61$ ms, and $T_{\text{avg rotation}} = 2.30$ ms.

A. *Best case:* In the optimal case, the blocks are mapped to contiguous sectors, on the same cylinder, that can be read one after the other without moving the head. Once the head is positioned over the first sector it takes two full rotations (5,000 sectors per rotation) of the disk to read all 10,000 blocks. So the total time to read the file is $T_{\text{avg seek}} + T_{\text{avg rotation}} + 2 \times T_{\text{max rotation}} = 6 + 2.30 + 9.22 = 17.52$ ms.

B. *Random case:* In this case, where blocks are mapped randomly to sectors, reading each of the 10,000 blocks requires $T_{\text{avg seek}} + T_{\text{avg rotation}}$ ms, so the total time to read the file is $(T_{\text{avg seek}} + T_{\text{avg rotation}}) \times 10{,}000 = 83{,}000$ ms (83 seconds!).

You can see now why it's often a good idea to defragment your disk drive!

Solution to Problem 6.5 (page 637)

This is a simple problem that will give you some interesting insights into the feasibility of SSDs. Recall that for disks, 1 PB = 10^9 MB. Then the following straightforward translation of units yields the following predicted times for each case:

A. Worst-case sequential writes (520 MB/s):

$$(10^9 \times 128) \times (1/520) \times (1/(86{,}400 \times 365)) \approx 7 \text{ years}$$

B. Worst-case random writes (205 MB/s):

$$(10^9 \times 128) \times (1/205) \times (1/(86{,}400 \times 365)) \approx 19 \text{ years}$$

C. Average case (50 GB/day):

$$(10^9 \times 128) \times (1/50{,}000) \times (1/365) \approx 6{,}912 \text{ years}$$

So even if the SSD operates continuously, it should last for at least 7 years, which is longer than the expected lifetime of most computers.

Solution to Problem 6.6 (page 640)

In the 10-year period between 2005 and 2015, the unit price of rotating disks dropped by a factor of 166, which means the price is dropping by roughly a factor of 2 every 18 months or so. Assuming this trend continues, a petabyte of storage, which costs about $30,000 in 2015, will drop below $200 after about eight of these factor-of-2 reductions. Since these are occurring every 18 months, we might expect a petabyte of storage to be available for $200 around the year 2027.

Solution to Problem 6.7 (page 644)

To create a stride-1 reference pattern, the loops must be permuted so that the rightmost indices change most rapidly.

```
1   int productarray3d(int a[N][N][N])
2   {
3       int i, j, k, product = 1;
4
5       for (j = N-1; j >= 0; j--) {
6           for (k = N-1; k >= 0; k--) {
7               for (i = N-1; i >= 0; i--) {
8                   product *= a[j][k][i];
9               }
10          }
11      }
12      return product;
13  }
```

This is an important idea. Make sure you understand why this particular loop permutation results in a stride-1 access pattern.

Solution to Problem 6.8 (page 645)

The key to solving this problem is to visualize how the array is laid out in memory and then analyze the reference patterns. Function `clear1` accesses the array using a stride-1 reference pattern and thus clearly has the best spatial locality. Function `clear2` scans each of the N structs in order, which is good, but within each struct it hops around in a non-stride-1 pattern at the following offsets from the beginning of the struct: 0, 12, 4, 16, 8, 20. So `clear2` has worse spatial locality than `clear1`. Function `clear3` not only hops around within each struct, but also hops from struct to struct. So `clear3` exhibits worse spatial locality than `clear2` and `clear1`.

Solution to Problem 6.9 (page 652)

The solution is a straightforward application of the definitions of the various cache parameters in Figure 6.26. Not very exciting, but you need to understand how the cache organization induces these partitions in the address bits before you can really understand how caches work.

Cache	m	C	B	E	S	t	s	b
1.	32	1,024	4	1	256	22	8	2
2.	32	1,024	8	4	32	24	5	3
3.	32	1,024	32	32	1	27	0	5

Solution to Problem 6.10 (page 660)

The padding eliminates the conflict misses. Thus, three-fourths of the references are hits.

Solution to Problem 6.11 (page 660)

Sometimes, understanding why something is a bad idea helps you understand why the alternative is a good idea. Here, the bad idea we are looking at is indexing the cache with the high-order bits instead of the middle bits.

A. With high-order bit indexing, each contiguous array chunk consists of 2^t blocks, where t is the number of tag bits. Thus, the first 2^t contiguous blocks of the array would map to set 0, the next 2^t blocks would map to set 1, and so on.

B. For a direct-mapped cache where $(S, E, B, m) = (512, 1, 32, 32)$, the cache capacity is 512 32-byte blocks with $t = 18$ tag bits in each cache line. Thus, the first 2^{18} blocks in the array would map to set 0, the next 2^{18} blocks to set 1. Since our array consists of only $(4{,}096 \times 4)/32 = 512$ blocks, all of the blocks in the array map to set 0. Thus, the cache will hold at most 1 array block at any point in time, even though the array is small enough to fit entirely in the cache. Clearly, using high-order bit indexing makes poor use of the cache.

Solution to Problem 6.12 (page 664)

The 2 low-order bits are the block offset (CO), followed by 3 bits of set index (CI), with the remaining bits serving as the tag (CT):

CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO
12	11	10	9	8	7	6	5	4	3	2	1	0

Solution to Problem 6.13 (page 664)

Address: `0x0D53`

A. Address format (1 bit per box):

CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO
0	1	1	0	1	0	1	0	1	0	0	1	1
12	11	10	9	8	7	6	5	4	3	2	1	0

B. Memory reference:

Parameter	Value
Cache block offset (CO)	`0x3`
Cache set index (CI)	`0x4`
Cache tag (CT)	`0x6A`
Cache hit? (Y/N)	N
Cache byte returned	—

Solution to Problem 6.14 (page 665)

Address: `0x0CB4`

A. Address format (1 bit per box):

CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO
0	1	1	0	0	1	0	1	1	0	1	0	0
12	11	10	9	8	7	6	5	4	3	2	1	0

B. Memory reference:

Parameter	Value
Cache block offset (CO)	`0x0`
Cache set index (CI)	`0x5`
Cache tag (CT)	`0x65`
Cache hit? (Y/N)	N
Cache byte returned	—

Solution to Problem 6.15 (page 665)

Address: `0x0A31`

A. Address format (1 bit per box):

CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO
0	1	0	1	0	0	0	1	1	0	0	0	1
12	11	10	9	8	7	6	5	4	3	2	1	0

B. Memory reference:

Parameter	Value
Cache block offset	0x1
Cache set index	0x4
Cache tag	0x51
Cache hit? (Y/N)	N
Cache byte returned	—-

Solution to Problem 6.16 (page 666)

This problem is a sort of inverse version of Practice Problems 6.12–6.15 that requires you to work backward from the contents of the cache to derive the addresses that will hit in a particular set. In this case, set 3 contains one valid line with a tag of 0x32. Since there is only one valid line in the set, four addresses will hit. These addresses have the binary form 0 0110 0100 11xx. Thus, the four hex addresses that hit in set 3 are

0x064C, 0x064D, 0x064E, and 0x064F

Solution to Problem 6.17 (page 672)

A. The key to solving this problem is to visualize the picture in Figure 6.48. Notice that each cache line holds exactly one row of the array, that the cache is exactly large enough to hold one array, and that for all i, row i of `src` and `dst` maps to the same cache line. Because the cache is too small to hold both arrays, references to one array keep evicting useful lines from the other array. For example, the write to `dst[0][0]` evicts the line that was loaded when we read `src[0][0]`. So when we next read `src[0][1]`, we have a miss.

`dst` array			`src` array		
	Col. 0	Col. 1		Col. 0	Col. 1
Row 0	m	m	Row 0	m	m
Row 1	m	m	Row 1	m	h

B. When the cache is 32 bytes, it is large enough to hold both arrays. Thus, the only misses are the initial cold misses.

`dst` array			`src` array		
	Col. 0	Col. 1		Col. 0	Col. 1
Row 0	m	h	Row 0	m	h
Row 1	m	h	Row 1	m	h

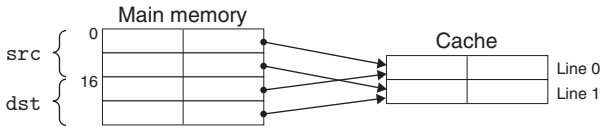

Figure 6.48
Figure for solution to Problem 6.17.

Solution to Problem 6.18 (page 673)

Each 32-byte cache line holds two contiguous `algae_position` structures. Each loop visits these structures in memory order, reading one integer element each time. So the pattern for each loop is miss, hit, miss, hit, and so on. Notice that for this problem we could have predicted the miss rate without actually enumerating the total number of reads and misses.

A. What is the total number of read accesses? 2,048 reads.

B. What is the total number of read accesses that miss in the cache? 1,024 misses.

C. What is the miss rate? $1024/2048 = 50\%$.

Solution to Problem 6.19 (page 674)

The key to this problem is noticing that the cache can only hold 1/2 of the array. So the column-wise scan of the second half of the array evicts the lines that were loaded during the scan of the first half. For example, reading the first element of `grid[8][0]` evicts the line that was loaded when we read elements from `grid[0][0]`. This line also contained `grid[0][1]`. So when we begin scanning the next column, the reference to the first element of `grid[0][1]` misses.

A. What is the total number of read accesses? 2,048 reads.

B. What is the total number of read accesses that hit in the cache? 1,024 misses.

C. What is the hit rate? $1024/2048 = 50\%$.

D. What would the hit rate be if the cache were twice as big? If the cache were twice as big, it could hold the entire `grid` array. The only misses would be the initial cold misses, and the hit rate would be $3/4 = 75\%$.

Solution to Problem 6.20 (page 674)

This loop has a nice stride-1 reference pattern, and thus the only misses are the initial cold misses.

A. What is the total number of read accesses? 2,048 reads.

B. What is the total number of read accesses that hit in the cache? 1,536 misses.

C. What is the hit rate? $1536/2048 = 75\%$.

D. What would the hit rate be if the cache were twice as big? Increasing the cache size by any amount would not change the miss rate, since cold misses are unavoidable.

Solution to Problem 6.21 (page 679)

The sustained throughput using large strides from L1 is about 12,000 MB/s, the clock frequency is 2,100 MHz, and the individual read accesses are in units of 16-byte `longs`. Thus, from this graph we can estimate that it takes roughly $2{,}100/12{,}000 \times 16 = 2.8 \approx 3.0$ cycles to access a word from L1 on this machine, which is roughly 1.25 times faster than the nominal 4-cycle latency from L1. This is due to the parallelism of the 4×4 unrolled loop, which allows multiple loads to be in flight at the same time.