1. Intro. We generate clauses that are satisfiable if and only if there's a sorting network on n lines for which (a) certain initial comparators are specified, and (b) all subsequent comparators can be done in t parallel steps.

We assume that $n < 16$, so that all lines can be identified by a single hexadecimal digit. Furthermore t is a single digit number, because 16 elements can be sorted in 9 rounds.

There are variables $ij\texttt{C}t$, denoting the existence of comparator $[i{:}j]$ at time t; $kj\texttt{B}t$, denoting internal nodes to check for interference/use of lines at time t; and $xi\texttt{V}t$, denoting the value on line i at time t when the input is x. (Here x is given as four hexadecimal digits.)

```
#define maxr 20    /* at most this many initial comparators */
#include <stdio.h>
#include <stdlib.h>
  int n, tt, ii, jj;    /* command-line parameters */
  int mask[maxr + maxr], m[16];
  int leaf[32];
  char needed[1 << 16];
  main(int argc, char *argv[])
  {
    register int i, is, j, js, k, l, r, t, x, y, z;
    ⟨ Process the command line 2 ⟩;
    ⟨ Generate the B and C clauses 3 ⟩;
    ⟨ Generate the V clauses 4 ⟩;
    ⟨ Generate the unit clauses 10 ⟩;
  }
```

2. ⟨ Process the command line 2 ⟩ ≡

```
  if (argc == 1 || (argc & 1) == 0 || argc > maxr + maxr + 3 ||
        sscanf(argv[1], "%d", &n) != 1 || sscanf(argv[2], "%d", &tt) != 1) {
    fprintf(stderr, "Usage: %s n t i1 j1 ... ir jr\n", argv[0]);
    exit(-1);
  }
  if (n < 3 || n > 15) {
    fprintf(stderr, "Sorry, n must be between 3 and 15!\n");
    exit(-2);
  }
  printf("~ sat-mintime-sort %d %d", n, tt);
  for (j = 3; argv[j]; j++) printf(" %s", argv[j]);
  printf("\n");
```

This code is used in section 1.

3. There are $n-1$ potential comparators that use line k, namely $[1{:}k]$, …, $[k{-}1{:}k]$, $[k{:}k{+}1]$, …, $[k{:}n]$. Place these at nodes $n-1$ through $2n-3$ of the complete binary tree with $n-1$ leaves. The $n-2$ internal nodes of this tree represent the condition that exactly one leaf in their subtree is true.

```
⟨Generate the B and C clauses 3⟩ ≡
  for (t = 1; t ≤ tt; t++)
    for (k = 1; k ≤ n; k++) {
      for (i = n − 1, j = 1; j ≤ n; j++) {
        if (j < k) leaf[i++] = (j ≪ 4) + k;
        else if (j > k) leaf[i++] = (k ≪ 4) + j;
      }
      for (j = n − 2; j; j−−) {
        if (j + j ≥ n − 1) printf("~%xC%d", leaf[j + j], t);
        else printf("~%x%xB%d", k, j + j, t);
        if (j + j + 1 ≥ n − 1) printf("␣~%xC%d\n", leaf[j + j + 1], t);
        else printf("␣~%x%xB%d\n", k, j + j + 1, t);
        if (j + j ≥ n − 1) printf("~%xC%d␣%x%xB%d\n", leaf[j + j], t, k, j, t);
        else printf("~%x%xB%d␣%x%xB%d\n", k, j + j, t, k, j, t);
        if (j + j + 1 ≥ n − 1) printf("~%xC%d␣%x%xB%d\n", leaf[j + j + 1], t, k, j, t);
        else printf("~%x%xB%d␣%x%xB%d\n", k, j + j + 1, t, k, j, t);
        printf("~%x%xB%d", k, j, t);
        if (j + j ≥ n − 1) printf("␣%xC%d", leaf[j + j], t);
        else printf("␣%x%xB%d", k, j + j, t);
        if (j + j + 1 ≥ n − 1) printf("␣%xC%d\n", leaf[j + j + 1], t);
        else printf("␣%x%xB%d\n", k, j + j + 1, t);
      }
    }
```

This code is used in section 1.

4.

```
⟨Generate the V clauses 4⟩ ≡
  ⟨Set up the masks 5⟩;
  ⟨Generate all the input cases 6⟩;
  for (x = 2; x < (1 ≪ n); x++)
    if (needed[x]) {
      for (t = 1; t ≤ tt; t++) {
        for (i = 1, is = 1 ≪ (n − 1); is; i++, is ≫= 1)
          for (j = i + 1, js = is ≫ 1; js; j++, js ≫= 1) ⟨Generate the clauses for i:j on x at time t 8⟩;
        for (i = 1, is = 1 ≪ (n − 1); is; i++, is ≫= 1)
          ⟨Generate the clauses for untouched line i on x at time t 9⟩;
      }
    }
```

This code is used in section 1.

5. ⟨ Set up the masks 5 ⟩ ≡

```
for (i = 0; argv[i + i + 3]; i++) {
  if (sscanf(argv[i + i + 3], "%d", &ii) ≠ 1 ∨ sscanf(argv[i + i + 4], "%d",
        &jj) ≠ 1 ∨ ii < 1 | ii > n | jj ≤ ii | jj > n) {
    fprintf(stderr, "Invalid␣comparator␣[%s:%s""]\n", argv[i + i + 3], argv[i + i + 4]);
    exit(−3);
  }
  mask[i] = ((1 ≪ n) ≫ ii) | ((1 ≪ n) ≫ jj);
}
r = i;
```

See also section 7.

This code is used in section 4.

6. (Minor trick here: A binary number x is already "sorted" if and only if $x \,\&\, (x+1)$ is zero.)

⟨ Generate all the input cases 6 ⟩ ≡

```
for (x = 2; x < (1 ≪ n); x++) {
  for (y = x, i = 0; i < r; i++) {
    t = mask[i] & −mask[i];
    if ((y & mask[i]) ≡ mask[i] − t) y ⊕= mask[i];
  }
  if (y & (y + 1)) needed[y] = 1;
}
```

This code is used in section 4.

7. There are six clauses, each of which should also include $\bar{C}^t_{ij}$: $(\bar{v}^t_i \vee v^{t-1}_i)$; $(\bar{v}^t_i \vee v^{t-1}_j)$; $(v^t_i \vee \bar{v}^{t-1}_i \vee \bar{v}^{t-1}_j)$; $(\bar{v}^t_j \vee v^{t-1}_i \vee v^{t-1}_j)$; $(v^t_j \vee \bar{v}^{t-1}_i)$; $(v^t_j \vee \bar{v}^{t-1}_j)$. We change v^t_i to x_i when $t = 0$ and to y_i when $t = tt$; and we omit clauses that are always true.

Let $m_i = 2^{n-i} - 1$. The first four of these clauses are always true when $x \le m[i]$, because $v^t_i = 0$ for all t in that case. Similarly, the second clause and the last three are always true when $(x+1) \,\&\, m[j-1] \equiv 0$, because $v^t_j = 1$ for all t in that case. These simplifications are important, because the SAT solver will not immediately discover them via unit propagation.

```
#define xi (x & is)
#define xj (x & js)
#define yi (y & is)
#define yj (y & js)
```

⟨ Set up the masks 5 ⟩ +≡

```
for (i = 0; i ≤ n; i++) m[i] = (1 ≪ (n − i)) − 1;
```

8. ⟨Generate the clauses for $i{:}j$ on x at time t 8⟩ ≡

```
{
  for (y = x; z = y & (y + 1); y −= z ≫ 1) ;    /* sort x */
  if (x ≤ m[i] ∨ (t ≡ 1 ∧ xi) ∨ (t ≡ tt ∧ ¬yi)) ;
  else {
    printf("~%x%xC%d", i, j, t);
    if (t ≠ tt) printf("␣~%04x%xV%d", x, i, t);
    if (t ≠ 1) printf("␣%04x%xV%d", x, i, t − 1);
    printf("\n");
  }
  if (x ≤ m[i] ∨ ((x + 1) & m[j − 1]) ≡ 0 ∨ (t ≡ 1 ∧ xj) ∨ (t ≡ tt ∧ ¬yi)) ;
  else {
    printf("~%x%xC%d", i, j, t);
    if (t ≠ tt) printf("␣~%04x%xV%d", x, i, t);
    if (t ≠ 1) printf("␣%04x%xV%d", x, j, t − 1);
    printf("\n");
  }
  if (x ≤ m[i] ∨ (t ≡ 1 ∧ (¬xi ∨ ¬xj)) ∨ (t ≡ tt ∧ yi)) ;
  else {
    printf("~%x%xC%d", i, j, t);
    if (t ≠ tt) printf("␣%04x%xV%d", x, i, t);
    if (t ≠ 1) {
      printf("␣~%04x%xV%d", x, i, t − 1);
      if ((x + 1) & m[j − 1]) printf("␣~%04x%xV%d", x, j, t − 1);
    }
    printf("\n");
  }
  if (x ≤ m[i] ∨ ((x + 1) & m[j − 1]) ≡ 0 ∨ (t ≡ 1 ∧ ¬xi) ∨ (t ≡ tt ∧ yj)) ;
  else {
    printf("~%x%xC%d", i, j, t);
    if (t ≠ tt) printf("␣%04x%xV%d", x, j, t);
    if (t ≠ 1) printf("␣~%04x%xV%d", x, i, t − 1);
    printf("\n");
  }
  if (((x + 1) & m[j − 1]) ≡ 0 ∨ (t ≡ 1 ∧ ¬xj) ∨ (t ≡ tt ∧ yj)) ;
  else {
    printf("~%x%xC%d", i, j, t);
    if (t ≠ tt) printf("␣%04x%xV%d", x, j, t);
    if (t ≠ 1) printf("␣~%04x%xV%d", x, j, t − 1);
    printf("\n");
  }
  if (((x + 1) & m[j − 1]) ≡ 0 ∨ (t ≡ 1 ∧ (xi ∨ xj)) ∨ (t ≡ tt ∧ ¬yj)) ;
  else {
    printf("~%x%xC%d", i, j, t);
    if (t ≠ tt) printf("␣~%04x%xV%d", x, j, t);
    if (t ≠ 1) {
      if (x > m[i]) printf("␣%04x%xV%d", x, i, t − 1);
      printf("␣%04x%xV%d", x, j, t − 1);
    }
    printf("\n");
  }
}
```

This code is used in section 4.

9. If i`1B`t is false, we have $v_i^t = v_i^{t-1}$.

⟨ Generate the clauses for untouched line i on x at time t 9 ⟩ ≡

```
{
  if (x ≤ m[i] ∨ ((x + 1) & m[i − 1]) ≡ 0 ∨ (t ≡ 1 ∧ xi) ∨ (t ≡ tt ∧ ¬yi)) ;
  else {
    printf("%x1B%d", i, t);
    if (t ≠ tt) printf("␣~%04x%xV%d", x, i, t);
    if (t ≠ 1) printf("␣%04x%xV%d", x, i, t − 1);
    printf("\n");
  }
  if (x ≤ m[i] ∨ ((x + 1) & m[i − 1]) ≡ 0 ∨ (t ≡ 1 ∧ ¬xi) ∨ (t ≡ tt ∧ yi)) ;
  else {
    printf("%x1B%d", i, t);
    if (t ≠ tt) printf("␣%04x%xV%d", x, i, t);
    if (t ≠ 1) printf("␣~%04x%xV%d", x, i, t − 1);
    printf("\n");
  }
}
```

This code is used in section 4.

10. Finally, we append unit clauses to suppress comparators that are redundant.

⟨ Generate the unit clauses 10 ⟩ ≡

```
for (i = 0; i < r; i++) {
  for (j = i + 1; j < r; j++) {
    if (mask[i] & mask[j]) break;
  }
  if (j ≡ r) {
    for (j = 1, t = 1 ≪ (n − 1); t; j++, t ≫= 1) {
      if (mask[i] & t) break;
    }
    for (k = j + 1, t ≫= 1; t; k++, t ≫= 1) {
      if (mask[i] & t) break;
    }
    printf("~%x%xC1\n", j, k);
  }
}
```

This code is used in section 1.

11. Index.

argc: 1, 2.
argv: 1, 2, 5.
exit: 2, 5.
fprintf: 2, 5.
i: 1.
ii: 1, 5.
is: 1, 4, 7.
j: 1.
jj: 1, 5.
js: 1, 4, 7.
k: 1.
l: 1.
leaf: 1, 3.
m: 1.
main: 1.
mask: 1, 5, 6, 10.
maxr: 1, 2.
n: 1.
needed: 1, 4, 6.
printf: 2, 3, 8, 9, 10.
r: 1.
sscanf: 2, 5.
stderr: 2, 5.
t: 1.
tt: 1, 2, 3, 4, 7, 8, 9.
x: 1.
xi: 7, 8, 9.
xj: 7, 8.
y: 1.
yi: 7, 8, 9.
yj: 7, 8.
z: 1.

⟨ Generate all the input cases 6 ⟩ Used in section 4.
⟨ Generate the B and C clauses 3 ⟩ Used in section 1.
⟨ Generate the V clauses 4 ⟩ Used in section 1.
⟨ Generate the clauses for $i{:}j$ on x at time t 8 ⟩ Used in section 4.
⟨ Generate the clauses for untouched line i on x at time t 9 ⟩ Used in section 4.
⟨ Generate the unit clauses 10 ⟩ Used in section 1.
⟨ Process the command line 2 ⟩ Used in section 1.
⟨ Set up the masks 5, 7 ⟩ Used in section 4.

SAT-MINTIME-SORT

	Section	Page
Intro	1	1
Index	11	6