TASK 1

To free memory of an inherited class one should use virtual destructor. That way a chain of destructors will be called instead of referred only. In this particular example, upon execution of `delete a;` only A's destructor will be called. The fix here is to declare A::~A() virtual to let call B::~B() and A::~A() afterward. Having said that, A's declaration should look like:

```
class A
{
public:
  A() {}
  virtual ~A() {}
};
```

Other code may be left as it is.
One shouldn't be worried with lack of `return 0;` statement in the end of `main`. This is allowed by standard and `return 0;` is implicit.

TASK 2

```
size_t compute_dirs_sizes(struct dir *d) {
  // Implement a recursive depth-first search-like walkthrough the directory tree
  size_t subdirsSize = 0;
  for (struct dir *subdir : d->dirs) {
    subdirsSize += compute_dirs_sizes(subdir);
  }

  size_t filesSize = 0;
  for (struct file *f : d->files) {
    filesSize = f->size();
  }

  // update directory node size
  d->size = subdirsSize + filesSize;
  return d->size;
}
```

Some notes. One could use a non-recursive way to traverse the directory tree.
Though such a method might require a lot more memory i.e. the stack should contain the whole tree while in recursive method only part of the tree resides in stack data structure.
Other, sophisticated, ways might be used here (e.g. multi-threaded ones with some use of std::future, etc), though they all imply DFS as for each node one should be aware of results for calculus on child nodes.

TASK 3

The scheduling problem here can be represented with a directed graph, where a node is a time-slot and an edge is present only for non-intersecting nodes (time slots). The edges are directed towards the node/time-slot which appears later in the schedule. Now the problem is to get length of the longest path in such a graph. Sure, neither node in the path is allowed to appear twice. As there's no need in retrieving the path itself a mere DFS might be used to get length of the longest path.

The graph will be represented as a ragged list i.e. vector of vectors where node X has edges to each node in V[X].
The graph is guaranteed to be acyclic.

To simplify the task here we sort schedule by ascending start time. This will take O(N * log(N)^2) for stable_sort.
Converting of schedule to graph will take O(N^2).
Generating list of roots uses O(N*log(N)) time, twice.
Complexity of DFS for single root is O(K), where K is number of subtree with given root.

This code isn't tested on use-cases. It only depicts general idea of the solution.

```
// makes lines shorter. no one loves long lines, right?
using SchedT = std::pair<int, int>;
using EdgesT = std::deque<size_t>;
using GraphT = std::vector<EdgesT>;

// get maximum depth in graph g from current node
size_t get_dfs_depth(const GraphT &g, size_t current) {
  // number of nodes is guaranteed to be at least 2
  size_t maxSubDepth = 0;

  if (!g[current].empty())
    for (size_t edgeTo : g[current]) {
      size_t depthFromTo = get_dfs_depth(g, edgeTo);
      if (depthFromTo > maxSubDepth)
        maxSubDepth = depthFromTo;
    }

  return 1 + maxSubDepth;
}

/// \param schedule represents the schedule for which a maximum number of teams
///                 should be calculated.
///                 Schedule is a vector of pairs. Each pair represents start and
///                 end time of time-slot. Each team has only one time-slot
///                 represented in schedule
/// \returns maximum number of teams that can appear in the schedule without
///          interfering each other
size_t task3(const std::vector<SchedT> &schedule) {
  // trivial case
  if (schedule.size() < 2)
    return schedule.size();

  // sort schedule on ascending start time
  std::stable_sort(schedule.begin(), schedule.end(),
                   [](const SchedT &lhs, const SchedT &rhs) {
                     return lhs.first < rhs.first;
                   });

  // Convert schedule to graph, node id is index of the slot in sorted schedule.
  // Get "roots" at the same time.
  // A "root" is a node, that's never referred to as an end of directed edge.
  std::vector<Edges> graph;
  std::set<size_t> roots;

  for (size_t id = 0; id < schedule.size(); ++id)
```

```
      roots.insert(id);

  graph.resize(schedule.size());

  // size of at least two is guaranteed by early bail out for trivial case
  for (size_t id1 = 0; id1 < schedule.size() - 1; ++id1) {
    EdgesT &edges = graph[id1];
    const SchedT &sc1 = schedule[id1];
    for (size_t id2 = id1 + 1; id2 < schedule.size(); ++id2) {
      // check for instersections
      if (sc1.second <= schedule[id2].first) {
        edges.push_back(id2);
        roots.erase(id2);
      }
    }
  }

  // initiate with 1 as even in worst case scenario we can
  // let one team to own the room
  size_t maxDepth = 1;

  for (size_t root : roots) {
    size_t depth = get_dfs_depth(graph, root);
    if (depth > maxDepth)
      depth = maxDepth;
  }

  return maxDepth;
}
```

TASK 4

Value of `val` isn't guaranteed after all the threads finish execution due access to the variable isn't atomic. To make access to `val` atomic one should use a mutex or make `val` atomic. There several ways to implement the function properly depending on technical needs. Though, the result will be the same, `val` will contain 5 * number of iterations in loop.

1: Use atomic variable.

```
std::atomic<size_t> val = 0;

void thread()
{
  for (size_t i = 0; i < 10000000; i++)
    val++;
}
```

2: Guard access with mutex. Lock mutex for the whole loop.

```
size_t val = 0;
std::mutex

void thread()
{
  mutex.lock(); // or use a lock guard
  for (size_t i = 0; i < 10000000; i++)
    val++;
  mutex.unlock();
```

```
}
```

3: Guard access with mutex. Lock mutex per each iteration of the loop.

```
size_t val = 0;
std::mutex

void thread()
{
  for (size_t i = 0; i < 10000000; i++) {
    mutex.lock(); // or use a lock guard
    val++;
    mutex.unlock();
  }
}
```

TASK 5

I assume that "sort" here is sort by ascending order. Now, a single swap will result in sorted vector if and only if two elements are misplaced. In other words permutation distance is only one.
Suppose, that the input vector contains only integers starting with 1, neither integer is skipped, i.e. {1, 3, 4} is illegal input vector. Time complexity is O(N), memory usage is constant.

The function will return true if a single swap will result in sorted vector. For sorted vector the function will return false as swap operation will its sorted state.

If any integer is skipped in the input vector, then a map should be created first to map input vector's objects into indices. For an arbitrary data type and additional compare predicate should be provided. Time complexity will be affected by creating a map and will look like O(N * log(N)) (provided comparison predicate is constant on time). Memory usage will be increased for map only.

```
bool task5(const std::vector<size_t> &v) {
  size_t diff = 0;
  // diff < 3 condition for early bail out
  for (size_t idx = 1; idx <= v.size() && diff < 3; ++v)
    if (idx != v[idx - 1])
        ++diff;

  return 2 == diff;
}
```

TASK 6

I'll use a straight-forward approach here. Find the first letter from input string on the board. Look for the second letter amongst neighbours. And so on. If the last letter is found, return true. If some letter can't be found, return false. Sure thing, use of letters (their positions) should be tracked.

```
using CoordT = std::pair<size_t, size_t>;
using UsageT = std::set<CoordT>;
using BoardT = std::vector<std::vector<char>>;

bool traverse(const BoardT &b, const CoordT &init, const std::string &word, size_t
idx, UsageT &usage, const size_t W, const size_t H) {
  const char letter = word[idx];
```

```
  auto traverse_sub = [&](const CoordT &check) -> bool {
    if (b[check.first][check.second] == letter) {
      // it was the last letter and the word is found
      if (word.length() - 1 == idx)
        return true;

      // otherwise, continue traversing
      usage.insert(check);

      if (traverse(b, check, word, idx + 1, usage, W, H))
        // the word is found
        return true;

      usage.erase(check);
    }

    return false;
  };

  // check neighbours for word[idx]
  if (init.first > 0)
    traverse_sub(b, std::make_pair(init.first - 1, init.second),
                 word, idx, usage, W, H);
  if (init.first < W - 1)
    traverse_sub(b, std::make_pair(init.first + 1, init.second),
                 word, idx, usage, W, H);
  if (init.second > 0)
    traverse_sub(b, std::make_pair(init.first, init.second – 1),
                 word, idx, usage, W, H);
  if (init.second < H - 1)
    traverse_sub(b, std::make_pair(init.first, init.second + 1),
                 word, idx, usage, W, H);
}

bool task6(const BoardT &board, const std::string &word) {
  // trivial cases
  if (word.empty())
    return true;

  if (board.empty())
    return false;

  const W = board.size();
  const H = board[0].size();

  if (!H)
    return false;

  if (W * H < word.length())
    return false;

  size_t x, y;
  UsageT usage;
  const char init = word[0];

  for (x = 0; x < W && !result; ++x)
    for (y = 0; y < H && !result; ++y)
      if (board[x][y] == init) {
        // trivial case
```

```
        if (word.length() == 1)
          return true;

        usage.clear();
        usage.insert(std::make_pair(x, y));
        if (traverse(board, std::make_pair(x, y), word, 1, usage, W, H))
          return true;
      }


  return false;
}
```

TASK 7

The following solution is the simplest one. It uses constant memory and O(n^2) time. One could use a quick-sort and binary search on sorted vector to make the function use less time (O(n * log(n) for both quick sort and search). On the other hand, worst-case scenario will require O(n^2) for quick-sort and O(n * log(n)) (logarithmic time for search for each element).

```
void task7(const std::vector<int> &v, int x) {
  if (v.size() < 2) {
    // no solutions
    printf("[]\n");
    return;
  }

  std::vector<std::pair<int, int>> result;

  for (size_t idx1 = 0; idx1 < v.size() - 1; ++idx1) {
    const int diff = x - v[idx1];

    // there're no equal elements in the input vector
    if (diff == v[idx1])
      continue;

    for (size_t idx2 = idx1 + 1; idx2 < v.size(); ++idx2)
      if (v[idx2] == diff)
        result.emplace_back(std::make_pair(v[idx1], v[idx2]));
  }

  // output
  printf("[");
  if (!result.empty()) {
    printf("[%i,%i]", result[0].first, result[1].second);

    for (size_t idx = 1; idx < result.size(); ++idx)
      printf(", [%i,%i]", result[idx].first, result[idx].second);
  }
  printf("]\n");
}
```