Algorithm Tips

# Time

1. **time\_t** t = time(0);
2. // number of second from 1970-1-1 0 o'clock.
3. **time\_t** rawtime;
4. time(&rawtime);
5. // t is equal to rawtime;
6. **char** tmp[64];
7. strftime(tmp, **sizeof**(tmp),
8. "%Y/%m/%d %X %A the %j day %z",
9. localtime(&t));
10. // 2018/06/16 20:51:44 Saturday the 167 day -0400
11. **struct** **tm** \* timeinfo;
13. timeinfo = localtime(&rawtime);
14. printf("Current local time and date: %s",
15. asctime(timeinfo));
16. // Current local time and date: Sat Jun 16 20:51:44 2018

# String

## String to Integer

1. // HEX string to integer
2. **int** value = 0;
3. std::string s = "7fffffff";
4. value = std::stoi(s, 0, 16);
5. s = "9ba2ef03";
6. // cannot use stoi, because out of range.
7. // value is 0x9ba2ef03 also -1683820797
8. value = std::stoul(s, 0, 16);
9. s = "9ba2ef03";
10. **size\_t** numLen = 0;
11. // numLen is 8
12. value = std::stoul(s, &numLen, 16);
13. s = "9ba 2ef03";
14. numLen = 0;
15. // numLen is 3; value is 0x000009ba
16. value = std::stoul(s, &numLen, 16);
17. // DEC string to integer
18. **int** value = 0;
19. std::string s = "2147483647";
20. value = std::stoi(s);
21. s = "2611146499";
22. // cannot use stoi, because out of range.
23. // value is 0x9ba2ef03 also -1683820797
24. value = std::stoul(s);
25. // uv is 2611146499;
26. unsigned **int** uv = std::stoul(s);
27. s = "2611146499";
28. **size\_t** numLen = 0;
29. // numLen is 10
30. value = std::stoul(s, &numLen);
31. s = "2611 146499";
32. numLen = 0;
33. // numLen is 4; value is 2611
34. value = std::stoul(s, &numLen);

## Integer to string

1. // Integer to HEX string
2. std::stringstream sstream;
3. sstream << std::hex << 0x123efb9a;
4. // result is "123efb9a"
5. std::string result = sstream.str();
7. sstream << std::hex << 0;
8. // result is "123efb9a0"
9. result = sstream.str();
11. // clear data in sstream
12. sstream.str("");
13. sstream << std::hex << 0;
14. // result is "0"
15. result = sstream.str();
16. // Integer to DEC string
17. // "2147483647"
18. std::string result = to\_string(INT\_MAX);
19. // "0"
20. result = to\_string(0);
21. // "-2147483648"
22. result = to\_string(INT\_MAX + 1);
23. // "2147483648"
24. result = to\_string(unsigned **int** (INT\_MAX + 1));
25. // "-2147483648"
26. result = to\_string(INT\_MIN);
27. // "123456"
28. result = to\_string(123456);

## Split string

1. vector<string> mySplit(**const** string & s, **const** string & delim) {
2. vector<string> res;
3. **int** last = 0;
4. **int** first = s.find(delim, last);
5. **while** (first != -1) {
6. res.push\_back(s.substr(last, first - last));
7. last = first + delim.size();
8. first = s.find(delim, last);
9. }
10. res.push\_back(s.substr(last, s.size() - last));
11. **return** res;
12. }
14. vector<string> mySplit(**const** string & s, **char** delim) {
15. vector<string> res;
16. **int** last = 0;
17. **int** first = s.find\_first\_of(delim, last);
18. **while** (first != -1) {
19. res.push\_back(s.substr(last, first - last));
20. last = first + 1;
21. first = s.find\_first\_of(delim, last);
22. }
23. res.push\_back(s.substr(last, s.size() - last));
24. **return** res;
25. }

## Match substring

### KMP

find x in s

1. **size\_t** findPatternKMP(**const** string & s, **const** string & x)
2. {
3. **if** (x.empty() || s.size() < x.size())
4. **return** -1;
5. // get prefix
6. vector<**int**> jump(x.size(), 0);
7. **int** i = 1, j = 0;
8. **while** (i < x.size()) {
9. **if** (x[i] == x[j]) {
10. jump[i] = j + 1;
11. ++j;
12. ++i;
13. }
14. **else** {
15. **if** (j == 0)
16. ++i;
17. **else**
18. j = jump[j - 1];
19. }
20. }
21. // match
22. j = 0;
23. i = 0;
24. **while** (i < s.size()) {
25. **if** (s[i] == x[j]) {
26. ++j;
27. ++i;
28. **if** (j == x.size())
29. **return** i - j;
30. }
31. **else** {
32. **if** (j != 0)
33. j = jump[j - 1];
34. **else**
35. ++i;
36. }
37. }
38. **return** -1;
39. }

### Robin Karp

1. **size\_t** RobinKarp(**const** string & s, **const** string & x)
2. {
3. **if** (x.empty() || s.size() < x.size())
4. **return** -1;
5. unsigned **char** pattern = 0;
6. unsigned **char** cmppattern = 0;
7. **for** (**int** i = 0; i < x.size(); ++i) {
8. pattern ^= x[i];
9. cmppattern ^= s[i];
10. }
11. cmppattern ^= s[x.size() - 1];
12. **for** (**int** i = 0; i <= s.size() - x.size(); ++i) {
13. cmppattern ^= s[i + x.size() - 1];
14. **if** (pattern == cmppattern) {
15. **int** match = **true**;
16. **for** (**int** j = 0; j < x.size(); ++j)
17. **if** (s[i + j] != x[j])
18. match = **false**;
19. **if** (match)
20. **return** i;
21. }
22. cmppattern ^= s[i];
23. }
24. **return** -1;
25. }

## Substring

1. string a("abcd");
2. string b = a.substr(0); // "abcd"
3. string c = a.substr(0, 3); // "abc"
4. string d = a.substr(2, 2); // "cd"
5. string e = a.substr(2, 4); // "cd"
6. string f = a.substr(4, 10); // ""
7. string g = a.substr(5, 10); // invalid string position

## Find inserted char O(log n)

Given one string s1, and then insert one char into this string at any place, to get s2, find the inserted char. O(log n) time.

1. **int** f(**const** string ori, **const** string mdf, **int** beg, **int** end) {
2. **int** mid = (end + beg) / 2;
3. **if** (end == beg) {
4. **return** end;
5. }
6. **if** (mdf[mid] != ori[mid]) {
7. **return** f(ori, mdf, beg, mid);
8. }
9. **else** {
10. **if** (mid > beg && mdf[mid] == ori[mid - 1]) {
11. **int** subres = f(ori, mdf, beg, mid - 1);
12. **if** (subres == mdf.size() - 1 || mdf[subres] != ori[subres]) {
13. **return** subres;
14. }
15. **return** f(ori, mdf, mid + 1, end);
16. }
17. **else** {
18. **return** f(ori, mdf, mid + 1, end);
19. }
20. }
21. }

# Container

notice: *vector*<int[10]> m; // do not do that

## Sorted containers (map, set, multiset)

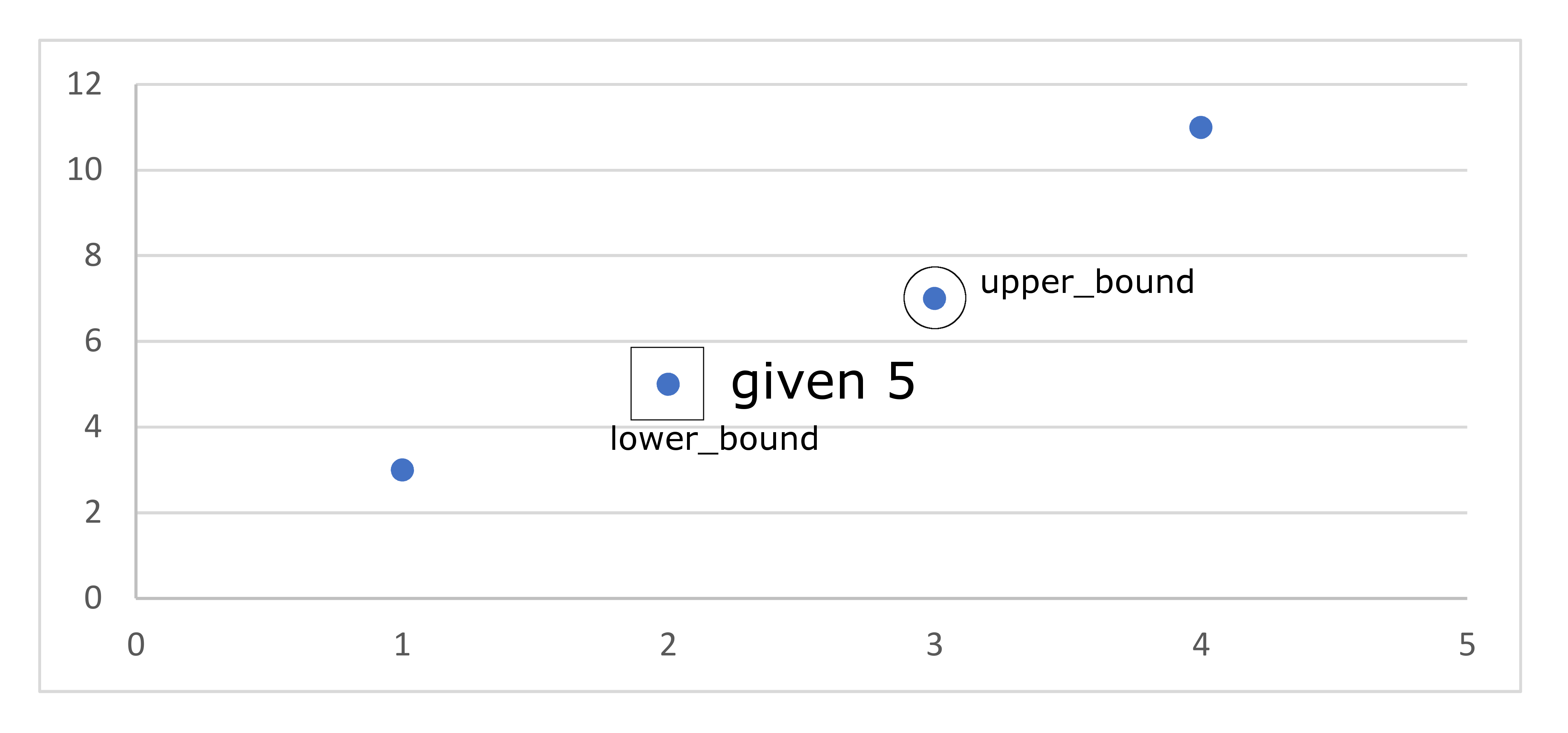
### common

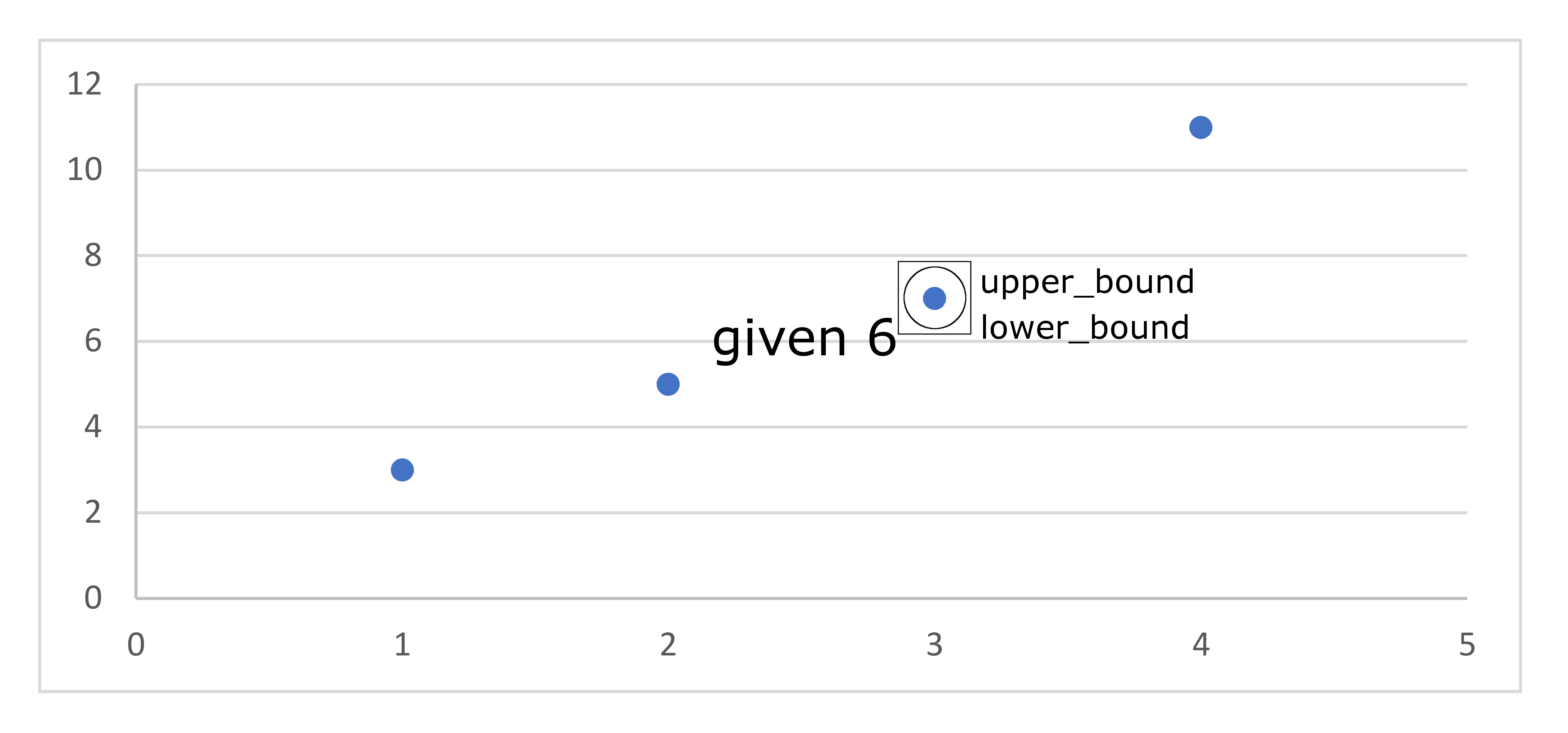
The elements in the container follow a strict order at all times. All inserted elements are given a position in this order.

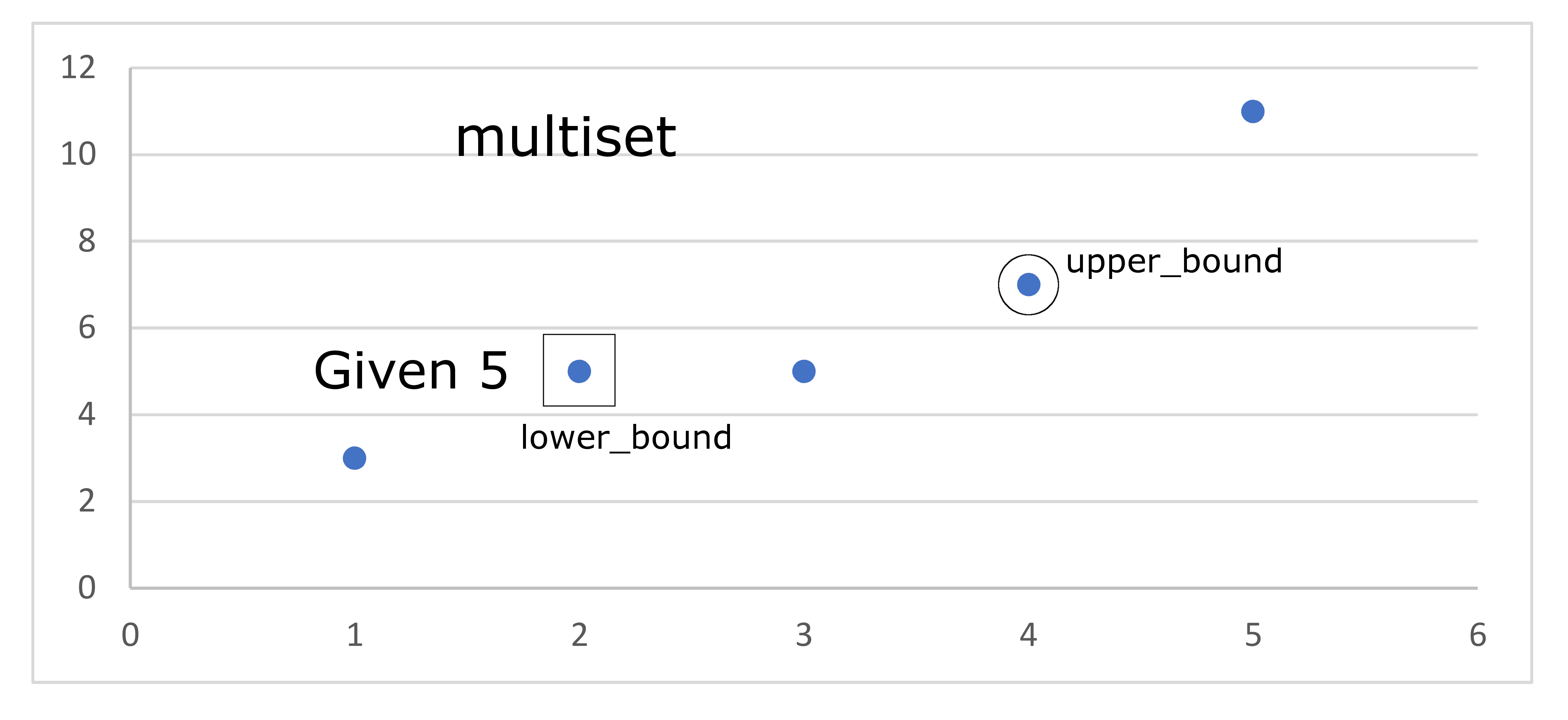
#### insert

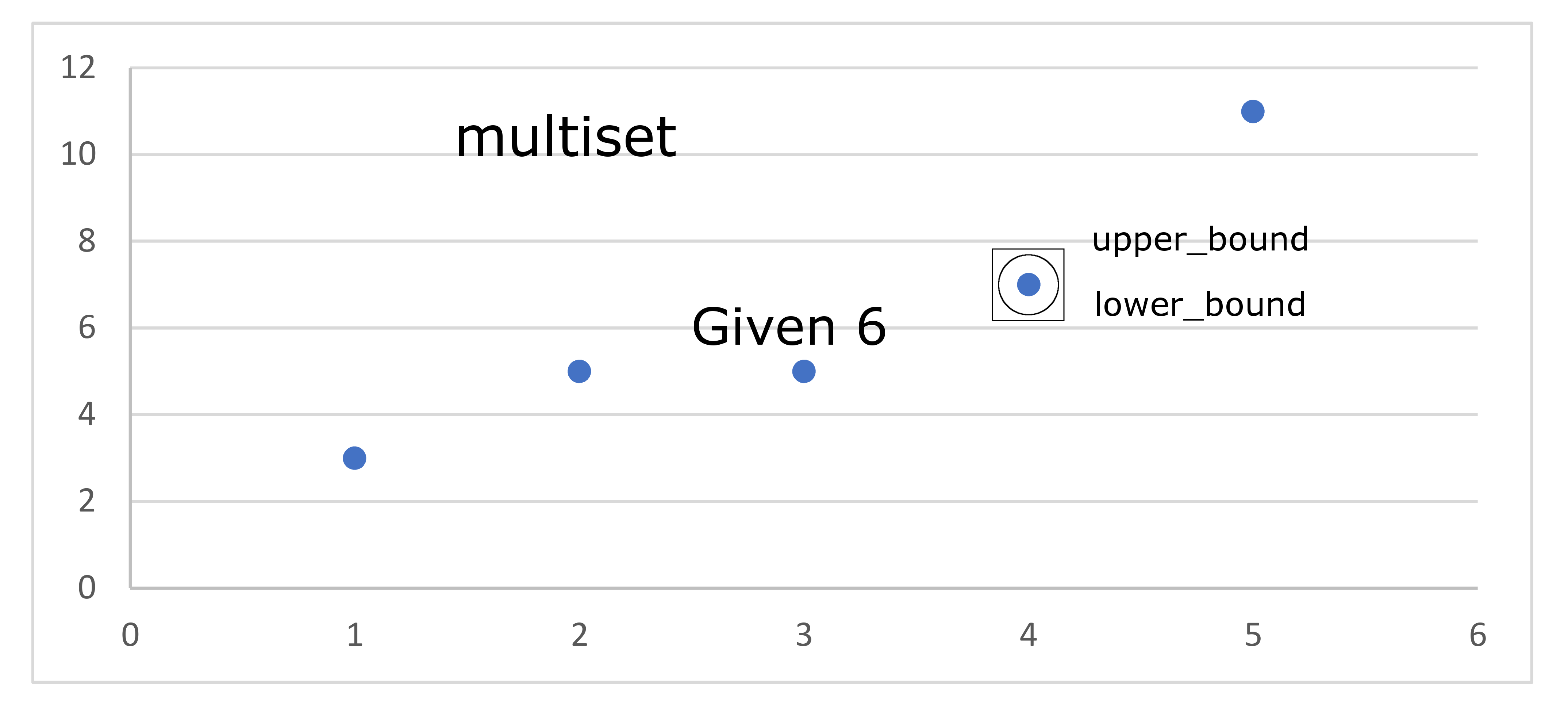
1. map<**int**, **int**> m;
2. vector<pair<**int**,**int**>> v;
3. v.emplace\_back(2, 57);
4. v.emplace\_back(1, 46);
5. v.emplace\_back(1, 49);
6. v.emplace\_back(3, 18);
7. // v: (2, 57);(1, 46);(1, 49);(3, 18)
8. m.insert(v.begin(), v.end());
9. // m: (1, 46);(2, 57);(3, 18)
11. set<**int**> s;
12. vector<**int**> vi;
13. vi.emplace\_back(2);
14. vi.emplace\_back(1);
15. vi.emplace\_back(1);
16. vi.emplace\_back(3);
17. // vi: (2);(1);(1);(3)
18. s.insert(vi.begin(), vi.end());
19. // m: (1);(2);(3)
21. multiset<**int**> ms;
22. vector<**int**> vm;
23. vm.emplace\_back(2);
24. vm.emplace\_back(1);
25. vm.emplace\_back(1);
26. vm.emplace\_back(3);
27. // vm: (2);(1);(1);(3)
28. ms.insert(vi.begin(), vi.end());
29. // ms: (1);(1);(2);(3)

#### upper\_bound and lower\_bound









# Tree

## In order traversal without Recursion

<https://www.youtube.com/watch?v=VsxLHGUqAKs>

1. **class** BTN {
2. **public**:
3. **int** val;
4. BTN \* m\_pLeft;
5. BTN \* m\_pRight;
6. BTN(**int** v):val(v),m\_pLeft(NULL),m\_pRight(NULL),
7. m\_pCurrent(NULL) {}
9. BTN \* getNext() {
10. **if** (m\_pCurrent == NULL) {
11. BTN \* ptr = **this**;
12. **while** (ptr) {
13. inorderstack.push\_back(ptr);
14. ptr = ptr->m\_pLeft;
15. }
16. m\_pCurrent = inorderstack.back();
17. inorderstack.pop\_back();
18. }
19. **else** {
20. BTN \* ptr = m\_pCurrent->m\_pRight;
21. **while** (ptr) {
22. inorderstack.push\_back(ptr);
23. ptr = ptr->m\_pLeft;
24. }
25. **if** (inorderstack.empty()) {
26. m\_pCurrent = NULL;
27. }
28. **else** {
29. m\_pCurrent = inorderstack.back();
30. inorderstack.pop\_back();
31. }
32. }
33. **return** m\_pCurrent;
34. }
36. **private**:
37. BTN \* m\_pCurrent;
38. list<BTN\*> inorderstack;
39. };

# Graph

## Topological order

1. list<string> getTopoOrder(vector<pair<string, string>> & dep)
2. {
3. map<string, **int**> depcnt;
4. map<string, list<string>> follow;
5. set<string> noindeg;
6. auto pd = dep.begin();
7. **while** (pd != dep.end()) {
8. auto prec = depcnt.find(pd->second);
9. **if** (prec == depcnt.end()) {
10. depcnt[pd->second] = 1;
11. }
12. **else** {
13. prec->second++;
14. }
15. noindeg.erase(pd->second);
16. **if** (depcnt.find(pd->first) == depcnt.end()) {
17. noindeg.insert(pd->first);
18. }
19. follow[pd->first].push\_back(pd->second);
20. ++pd;
21. }
22. list<std::string> res;
23. **while** (!noindeg.empty()) {
24. string prer = \*noindeg.begin();
25. res.push\_back(prer);
26. noindeg.erase(noindeg.begin());
27. auto pf = follow.find(prer);
28. **if** (pf != follow.end()) {
29. auto pff = pf->second.begin();
30. **while** (pff != pf->second.end()) {
31. depcnt[\*pff]--;
32. **if** (depcnt[\*pff] == 0) {
33. noindeg.insert(\*pff);
34. depcnt.erase(\*pff);
35. }
36. ++pff;
37. }
38. }
39. }
40. **return** res;
41. }

# Math

## Prime factorization

1. vector<**int**> primeFactorization(**int** num) {
2. vector<**int**> res;
3. **int** start = 2;
4. **while** (num > 1) {
5. **bool** find = **false**;
6. **for** (**int** i = start; i \* i <= num; ++i) {
7. **if** (num % i == 0) {
8. start = i;
9. res.push\_back(i);
10. num /= i;
11. find = **true**;
12. **break**;
13. }
14. }
15. **if** (!find)
16. **break**;
17. }
18. res.push\_back(num);
19. **return** res;
20. }