**105 STL Algorithms**

1. make\_heap(begin(vectorName), end(vectorName)
2. push\_back(element)
3. push\_heap(begin(heapName), end(heapName)) – To insert element as heap
4. pop\_heap(begin(heapName), end(headName)) : to remove element
5. pop\_back() : lastNumber returned.
6. sort\_heap()
7. sort(begin(colName), end(colName))
8. partial\_sort(begin(colName), end(colName))
9. nth\_element(begin(colName), end(colName))

inplace\_merge – combines two collection to sorted

partition\_point()

rotate(collection)

shuffle(collection)

next\_permutation(begin(), end())

prev\_permutation(begin(), end())

reverse()

stable\_sort() : // stable\_ keeps the relative order.

stable\_partition()

is\_sorted()

is\_partitioned()

is\_heap()

is\_sorted\_until() -> returns iterator to the position at which ends.

is\_partitioned\_until()

is\_heap\_until()

count()

accumulate()/ (transform\_) reduce

partial\_sum()

(transform\_)inclusive\_scan

(transform\_)exclusive\_scan

inner\_product()

adjacent\_difference()

sample()

all\_of() -> empty collection -> true

any\_of() -> empty collection -> false

none\_of() -> empty collection -> true

equal()

lexicographical\_compare()

is\_permutation()

mismatch() -> return pair<iterator, iterator>

find(begin(), end(), value)

adjacent\_find() -> first positions where two values are adjacent along with v

equal\_range() -> search in sorted collection

lower\_bound() ->

upper\_bound() ->

binary\_search(begin(), end(), val) -> bool

search(begin(), end(), subrange)

find\_end(begin(), end(), subrange) -> search from end

find\_first\_of()

max\_element()

min\_element()

minmax\_element()

set Algos

sorted vectors are called sets

set\_difference(begin(a), end(a), begin(b), end(b), std::back\_inserter(results)) : returns A-B linear complexity.

set\_intersection:

set\_union

set\_symmetric\_difference()

include

merge

copy(colName, last, outItr) -> outItr can be back\_inserter

move(first, last, out)

swap\_ranges(first, end, out)

copy\_backward

move\_backward

fill(first, last, 43) -> put same values everywhere

generate(first, last, f) ->fill the value with function f

iota(first, last, 43) -> incrementing value is filled in the coll.

replace(first, end, 43, 44) -> replace every 43 with 44

remove(begin(), end(), 99) -> pull up collection. Returns a itr.

coll.erase(begin(), end())

*erase(collection, 99)*

unique(begin(), end()): removes duplicate adjacent values

remove\_copy()

unique\_copy()

reverse\_copy()

rotate\_copy()

replace\_copy()

partition\_copy()

partial\_sort\_copy()

find\_if()

find\_if\_not()

count\_if()

remove\_if()

remove\_copy\_if()

replace\_if()

replace\_copy\_if()

copy\_if()

transform(begin(), end(), back\_inserter(results), f)-> function

transform(begin(), end(), begin(Second), back\_inserter(results), function)

for\_each(begin(), end(), function) => applies the function to every element.

Does not cares about the return value of function.

Performs side-effects.

fill() // all three have = operator and works upon the data which is already

copy() // created

move()

uninitialized\_fil

uninitialized\_copy

unitialized\_move

example:

uninitialized\_fill(first, end, 42);

destroy(first, end)

unitialized\_default\_construct

uninitialized\_value\_construct

copy\_n

fill\_n(begin(), n , value) : fills till n value

variations: fill(back\_inserter(coll), 5,43)

generate\_n

search\_n

for\_each\_n

unitialized\_copy\_n

boost\_\_algorithms