Project 2
Christian Mendez, email: christianmendez@csu.fullerton.edu
Michelle Nguyen, email: michellenguyen12@csu.fullerton.edu

Exhaustive Optimizatoin Algorithm

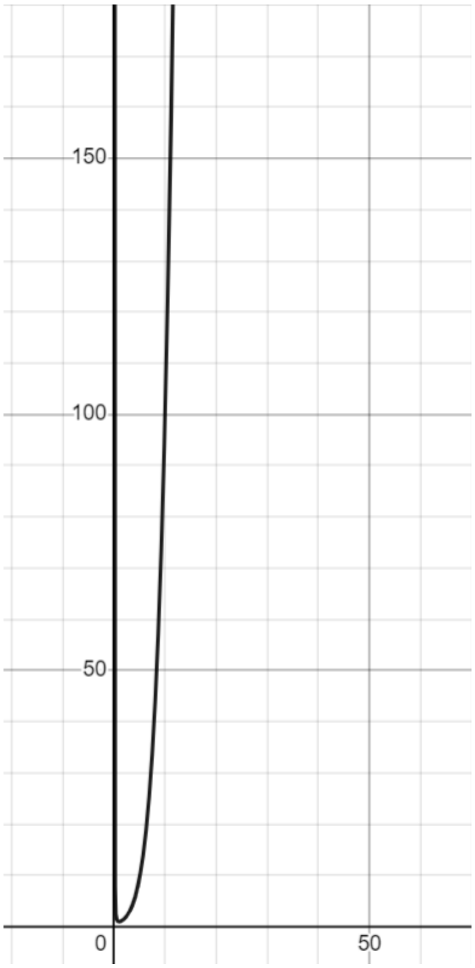
pseudo code:

```
def crane_unloading_exhaustive(setting):
        assert(setting.rows() > 0)
        assert(setting.columns() > 0)
        const size_t max_steps = setting.rows() + setting.columns() - 2
        assert(max_steps < 64)
        path best(setting)

        for (size_t steps = 1; steps <= max_steps; ++steps) :
                for (size_t bits = 0; bits <= pow(2,steps)-1; ++bits) :
                        path curr(setting)
                        bool valid = true

                        for (size_t k = 0; k < steps; ++k):
                                int bit = (bits >> k) & 1

                                if (bit == 1) :
                                        if (curr.is_step_valid(STEP_DIRECTION_EAST)) :
                                                curr.add_step(STEP_DIRECTION_EAST
                                        else valid = false
                                else :
                                         if (curr.is_step_valid(STEP_DIRECTION_SOUTH)) :
                                                curr.add_step(STEP_DIRECTION_SOUTH)
                                        else valid = false
                        endfor
        if (valid && (curr.total_cranes() > best.total_cranes())) :
                best = curr

                endfor
        endfor
 return best
```

Step count:
Sc = 1 + max(0,1) + max(0,1)
Sc = 3 + 2 + max(1,0)
Sc = 5 + max(1,0)
Sc = 6 + n * log(n) * n
Sc = 6 + n^2logn

Proof:
Show that 6 + n^2logn belongs to O(n^2logn)
F(n) = 6 + n^2logn
G(n) =n^2logn
Using def. F(n) <= c*G(n), n>=n0
By def. 6 + n^2logn <=c* n^2logn, n>n0
Let c = 16 and n0 = 1
6 + n^2logn <= 16*(1)^2log(1)
7 <= 16
This is true, hence 6 + n^2logn belongs to the O(n^2logn) time complexity

Graph:



<u>**Dynamic Programming Algorithm**</u>

Pseudo Code:

```
def crane_unloading_dyn_prog(const grid& setting):
        assert(setting.rows() > 0)
        assert(setting.columns() > 0)

        using cell_type = std::optional<path>

        std::vector<std::vector<cell_type>>
        A(setting.rows(),std::vector<cell_type>(setting.columns()))

         A[0][0] = path(setting)
         assert(A[0][0].has_value())

         for (coordinate r = 0; r < setting.rows(); ++r) :
                for (coordinate c = 0; c < setting.columns(); ++c) :
```

```
                if (setting.get(r, c) != CELL_BUILDING):
                        cell_type from_above = std::nullopt
                         cell_type from_left = std::nullopt

                         if (r > 0 && A[r-1][c].has_value()):
                                from_above = A[r-1][c]

                                if (from_above->is_step_valid(STEP_DIRECTION_SOUTH)):
                                        from_above->add_step(STEP_DIRECTION_SOUTH)

                        if (c > 0 && A[r][c-1].has_value()):
                                from_left = A[r][c-1];

                        if (from_left->is_step_valid(STEP_DIRECTION_EAST)):
                                from_left->add_step(STEP_DIRECTION_EAST)

                        if (from_above.has_value() && from_left.has_value()):
                                if (from_above->total_cranes() > from_left->total_cranes()):
                                        A[r][c] = from_above
                                Else:   A[r][c] = from_left

                         if (from_above.has_value() && !(from_left.has_value())):
                                A[r][c] = from_above

                         if (from_left.has_value() && !(from_above.has_value())):
                                A[r][c] = from_left

        endfor
endfor

 cell_type* best = &(A[0][0])
 assert(best->has_value())

 for (coordinate r = 0; r < setting.rows(); ++r):
        for (coordinate c = 0; c < setting.columns(); ++c):
                if (A[r][c].has_value() && A[r][c]->total_cranes() > (*best)->total_cranes()) :
                        best = &(A[r][c])
        endfor
 endfor

 assert(best->has_value())

 return **best
```

Step count:

Sc = 3+3+3+2+2+n[n*(2+max(1+8+7+5+5,0))]+3+5n^2

Sc=16+n[n*(2+35))]+5n^2

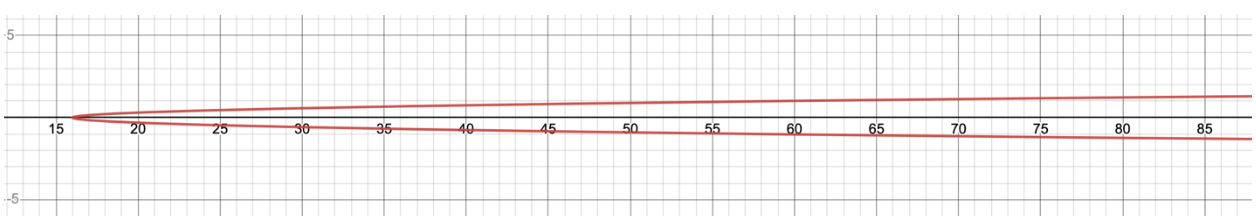
Sc=16+42n^2

Step Count = 42n^2+16

Proof:

Show that 42n^2+16 belongs to O(n^2)
F(n) =42n^2+16
G(n) =n^2
Using def. F(n) <= c*G(n), n>=n0
By def. 42n^2+16 <=c* n^2, n>n0
Let c = 60 and n0=1
42(1)^2+16<= 60*(1)^2
58<=60
This is true, hence 42n^2+16 belongs to the O(n^2) time complexity

Graph:



Questions:

1. Is there a noticeable difference in the performance of the two algorithms? Which is faster, and by how much? Does this surprise you?

There is a noticeable difference between the two algorithms because the dynamic programming algorithm ran quicker than the exhaustive optimization algorithm. The faster algorithm for our case would be the dynamic programming algorithm since the exhaustive optimization algorithm runs at a time complexity of O(n^2logn), whereas the dynamic programming algorithm runs at a time complexity of O(n^2). O(n^2logn) is slower than O(n^2) by quite a bit. For example, if you consider n as the size of 1000, O(n^2) would become 1,000,000, whereas O(n^2logn) would be at 3,000,000, therefore log n grows at a slower rate than n^2. This doesn't surprise me because the dynamic programming algorithm is supposed to be faster than the exhaustive optimization algorithm.

2. Are your empirical analyses consistent with your mathematical analyses? Justify your answer.

Our empirical analysis is consistent with our mathematical analysis because when we ran the experiment, the dynamic programming was quicker and more efficient than the exhaustive search algorithm. Our dynamic programming algorithm was able to solve the same problem as the exhaustive search algorithm at a time complexity of O(n^2).

3. Is this evidence consistent or inconsistent with hypothesis 1? Justify your answer.

This evidence is consistent with our hypothesis 1 because the dynamic programming algorithm ran faster than the exhaustive optimization algorithm. Similarly, to our previous answers, the dynamic programming algorithm had a time complexity of O(n^2) which is faster than the exhaustive optimization algorithm with a time complexity of O(n^2logn).