Chapter 3

Optimization I:
Brute force and Greedy strategy

A generic definition of an optimization problem involves a set of constraints that defines a subset in some underlying space (like the Euclidean space $\mathbb{R}^n$) called the feasible subset and an objective function that we are trying to maximize or minimize as the case may be over the feasible set. A very important common optimization problem is Linear Programming where there is a finite set of linear constraints and the objective function is also linear - the underlying space is Euclidean. A convex function $f$ satisfies $f(\lambda \cdot x + (1 - \lambda) \cdot y) \leq \lambda f(x) + (1 - \lambda) f(y)$ where $0 < \lambda < 1$. A convex programming problem is one where the objective function is convex and so is the feasible set. Convex programming problem over Euclidean real space have a nice property that local optima equals the global optimum. Linear programming falls under this category and there are provably efficient algorithms for linear programming.

Many important real life problems can be formulated as optimization problems and therefore solving them efficiently is one of the most important area of algorithm design.

3.1 Heuristic search approaches

In this section, we will use the knapsack problem as the running example to expose some of the algorithmic ideas. The 0-1 Knapsack problem is defined as follows.

Given a knapsack of capacity $C$ and $n$ objects of volumes $\{w_1, w_2 \ldots w_n\}$ and profits $\{p_1, p_2 \ldots p_n\}$, the objective is to choose a subset of $n$ objects that fits into the knapsack and that maximizes the total profit.

In more formal terms, let $x_i$ be 1 if object $i$ is present in the subset and 0 otherwise.
The knapsack problem can be stated as

\[
\text{Maximize } \sum_{i=0}^{n} x_i \cdot p_i \text{ subject to } \sum_{i=0}^{n} x_i \cdot w_i \leq C
\]

Note that the constraint \( x_i \in \{0,1\} \) is not linear. A simplistic approach will be to enumerate all subsets and select the one that satisfies the constraints and maximizes the profits. Any solution that satisfies the capacity constraint is called a feasible solution. The obvious problem with this strategy is the running time which is at least \( 2^n \) corresponding to the power-set of \( n \) objects.

We can imagine that the solution space is generated by a binary tree where we start from the root with an empty set and then move left or right according to selecting \( x_1 \). At the second level, we again associate the left and right branches with the choice of \( x_2 \). In this way, the \( 2^n \) leaf nodes correspond to each possible subset of the power-set which corresponds to a \( n \) length 0-1 vector. For example, a vector \( 000\ldots1 \) corresponds to the subset that only contains \( x_n \).

Any intermediate node at level \( j \) from the root corresponds to partial choice among the objects \( x_1, x_2 \ldots x_j \). As we traverse the tree, we keep track of the best feasible solution among the nodes visited - let us denote this by \( T \). At a node \( v \), let \( S(v) \) denote the subtree rooted at \( v \). If we can estimate the maximum profit \( P(v) \) among the leaf nodes of \( S(v) \), we may be able to prune the search. Suppose \( L(v) \) and \( U(v) \) are the lower and upperbounds of \( P(v) \), i.e. \( L(v) \leq P(v) \leq U(v) \). If \( U(v) < T \), then there is no need to explore \( S(v) \) as we cannot improve the current best solution, viz., \( T \). In fact, it is enough to work with only the upper-bound of the estimates and \( L(v) \) is essentially the current partial solution. As we traverse the tree, we also update \( U(v) \) and if it is less than \( T \), we do not search the subtree any further. This method of pruning search is called branch and bound and although it is clear that there it is advantageous to use the strategy, there may not be any provable savings in the worst case.

**Exercise 3.1** Construct an instance of a knapsack problem that visits every leaf node, even if you use branch and bound. You can choose any well defined estimation.

**Example 3.1**

Let the capacity of the knapsack be 15 and the weights and profits are respectively

<table>
<thead>
<tr>
<th>Profits</th>
<th>10</th>
<th>10</th>
<th>12</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volume</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

We will use the ratio of profit per volume as an estimation for upperbound. For the above objects the ratios are 5, 2.5, 2 and 2. Initially, \( T = 0 \) and \( U = 5 \times 15 = 75 \). After including \( x_1 \), the residual capacity is 13 and \( T = 10 \). By proceeding this way, we
obtain $T = 38$ for $\{x_1, x_2, x_4\}$. By exploring further, we come to a stage when we have included $x_1$ and decided against including $x_2$ so that $L(v) = 10$, and residual capacity is 13. Should we explore the subtree regarding $\{x_3, x_4\}$? Since profit per volume of $x_3$ is 2, we can obtain $U(v) = 2 \times 13 + 10 = 36 < L = 38$. So we need not search this subtree. By continuing in this fashion, we may be able to prune large portions of the search tree. However, it is not possible to obtain any provable improvements.

### 3.1.1 Game Trees *

A *minmax* tree is used to represent a game between two players who alternately make moves trying to win the game. We will focus on a special class of minimax trees called *AND-OR* trees where alternate levels of trees are labelled as OR nodes starting with the root node and the remaining are labelled AND nodes. Let 1 represent a win for player 1 and 0 represent a loss for player 1 who makes the first move at the root - the numbers are flipped for player 2. The leaf nodes correspond to the final state of the game and are labelled 1 or 0 corresponding to win or loss for player 1. We want to compute the values of the intermediate nodes using the following criteria. An OR node has value 1 if one of the children is 1, and 0 otherwise - so it is like the boolean function $OR$. An AND node behaves like a boolean AND function - it is 0 if one of the children is 0. The interpretation is as follows - the player at the root can choose any of the branches that leads to a win. However at the next level, he is at the mercy of the other player - only when both branches for the other player leads to a win (for the root), the root will win, otherwise the other player can inflict a loss.

For concreteness, we will consider game trees where each internal node has two children. So the evaluation of this Game Tree works as follows. Each leaf node is labelled 0 or 1 and an internal node as AND or OR - these will compute the boolean function of the value of the two child nodes. The value of the game tree is the value available at the root node.

First consider a single level AND tree that evaluates to 0. If we use a fixed order to inspect the leaf nodes, in the worst case, both leaf nodes may have to be searched if there is one 0 and it is the second branch. On the other hand, if we randomly choose the order, for an answer 0, with probability $1/2$, we will end up searching only one node (the one labelled 0) and we do not have to evaluate the other one. Therefore the expected number of look-ups is $3/2$ for an AND node with answer 0 and the same holds for an OR node with answer 1. For the other cases, there is no saving. However any interesting game tree will have at least two levels, one AND and the other OR. Then you can see that for an AND node to be 1, both the child OR nodes must be 1 which is the good case for OR.

In essence, we are applying the branch-and-bound method to this problem, and we obtain a provable improvement in the following way. The two children are evaluated
in a random order.

Consider a tree with depth $2k$ (i.e. $4^k$ leaf nodes) with alternating AND and OR nodes, each type having $k$ levels. We will show that the expected cost of evaluation is $3^k$ by induction on $k$.

**Exercise 3.2** Show that for $k = 1$, the expected number of evaluations is 3. (You must consider all cases of output and take the worst, since we are not assuming any distribution on input or output).

Let us consider a root with label OR and its two AND children, say $y$ and $z$, whose children are OR nodes with $2(k - 1)$ depth. We have the two cases

**output is 0 at the root** Both $y$ and $z$ must evaluate to 0. Since these are AND nodes, again with probability 1/2, we will end up evaluating only one of the children (of $y, z$) that requires expected $\frac{1}{2} \cdot (1 + 2) \cdot 3^{k-1} = \frac{3}{2} \cdot 3^{k-1}$ steps for $y$ as well as $z$ from Induction hypothesis. This adds up to a total of expected $2 \cdot \frac{3}{2} \cdot 3^{k-1} = 3^k$ steps for $y$ and $z$.

**Output is 1 at the root** At least one of the AND nodes $y, z$ must be 1. With probability 1/2 this will be chosen first and this can be evaluated using the expected cost of evaluating two OR nodes with output 1. By induction hypothesis this is $2 \cdot 3^{k-1}$.

The other possibility (also with probability 1/2) is that the first AND node (say $y$) is 0 and the second AND node is 1. The expected cost of the first AND node with 0 output is $1/2 \cdot 3^{k-1} + 1/2 \cdot (3^{k-1} + 3^{k-1})$ - the first term corresponds to the scenario that the first child evaluates to 0 and the second term corresponds to evaluating both children of $y$ are evaluated. The expected cost of evaluating $y$ having value 0 is $3/2 \cdot 3^{k-1}$.

The expected number of evaluation for second AND node $z$ with output 1 is $2 \cdot 3^{k-1}$ since both children must be evaluated.

So the total expected cost is $1/2 \cdot 3^{k-1}(2 + 3/2 + 2) = 2.75 \cdot 3^{k-1} < 3^k$.

In summary, for an OR root node, regardless of the output, the expected number of evaluations is bounded by $3^k$.

**Exercise 3.3** Establish a similar result for the AND root node.

If $N$ the number of leaf nodes, then the expected number of evaluations is $N^{\log_{4}3} = N^{\alpha}$ where $\alpha < 0.8$. 
3.2 A framework for Greedy Algorithms

There are very few algorithmic techniques for which the underlying theory is as precise and clean as what we will discuss here. Let us define the framework. Let $S$ be a set and $M$ be a subset \(^1\) of $2^S$. Then $(S, M)$ is called a subset system if it satisfies the following property

For all subsets $T \in M$, for any $T' \subset T$, $T' \in M$

Note that the empty subset $\emptyset \in M$. The family of subsets $M$ is often referred to as independent subsets and one may think of $M$ as the feasible subsets.

Example 3.2 For the maximal spanning tree problem on a graph $G = (V, E)$, $(E, F)$ is a matroid where $F$ is the set of all subgraphs without cycles (i.e. all the forests).

For any weight function $w : S \rightarrow \mathbb{R}^+$, the optimization problem is defined as finding a subset from $M$ for which the cumulative weight of the elements is maximum among all choices of subsets from $M$. A simple way to construct a subset is the following greedy approach.

\begin{algorithm}
\textbf{Algorithm Gen\_Greedy}

Let $e_1, e_2 \ldots e_n$ be the elements of $S$ in decreasing order of weights. Initialize $T = \emptyset$.

For $i = 1$ to $n$ do

- In the $i$-th stage
  - If $T \cup \{e_i\} \in M$, then $T \leftarrow T \cup \{e_i\}$

Output $T$ as the solution

\end{algorithm}

The running time of the algorithm is dependent mainly on the test for independence which depends on the specific problem. $M$ is not given explicitly as it may be very large (even exponential\(^2\)). Instead, a characterization of $M$ is used to perform the test.

What seems more important is the question - Is $T$ the maximum weight subset?

This is answered by the next result

Theorem 3.1 The following are equivalent

\(^1M\) is a family of subsets of $S$

\(^2\)The number of spanning trees of a complete graph is $n^{n-2}$
1. Algorithm Gen_Greedy outputs the optimal subset for any choice of the weight function. Note that in this case the subset system is called a matroid.

2. **exchange property**
   For any \( s_1, s_2 \in M \) where \(|s_1| < |s_2|\), then there exists \( e \in s_2 - s_1 \) such that \( s_1 \cup \{e\} \in M \).

3. For any \( A \subset S \), all maximal subsets of \( A \) have the same cardinality. A maximal subset \( T \) of \( A \) implies that there is no element \( e \in A - T \) such that \( T \cup \{e\} \in M \).

The obvious use of the theorem is to establish properties 2 or 3 to justify that a greedy approach works for the problem. On the contrary, we can try to prove that one of the properties doesn’t hold (by a suitable counterexample), then greedy cannot always return the optimum subset.

**Proof:** We will prove it in the following cyclic implications - Property 1 implies Property 2. Then Property 2 implies Property 3 and finally Property 3 implies Property 1.

**Property 1 implies Property 2** We will prove it by contradiction. Suppose Property 2 doesn’t hold for some subsets \( s_1 \) and \( s_2 \). That is, we cannot add any element from \( s_2 - s_1 \) to \( s_1 \) and keep it independent. Further, wlog, let \(|s_2| = p + 1\) and \(|s_1| = p\).

Let us define a weight function on the elements of \( S \) as follows

\[
  w(e) = \begin{cases} 
    p + 2 & \text{if } e \in s_1 \\
    p + 1 & \text{if } e \in s_2 - s_1 \\
    0 & \text{otherwise}
  \end{cases}
\]

The greedy approach will pick up all elements from \( s_1 \) and then it won’t be able to choose any element from \( s_2 - s_1 \). The greedy solution has weight \((p+2)|s_1| = (p+2)\cdot p\).

By choosing all elements of \( s_2 \), the solution has cost \((p+1)\cdot(p+1)\) which has a higher cost than greedy and hence it is a contradiction of Property 1 that is assumed to be true.

**Property 2 implies Property 3** If two maximal subsets of a set \( A \) have different cardinality, it is a violation of Property 2. Since both of these sets are independent, we should be able augment the set \( s_1 \) with an element from \( s_2 \).

**Property 3 implies Property 1** Again we will prove by contradiction. Let \( e_1 e_2 \ldots e_i \ldots e_n \) be the edges chosen by the greedy algorithm in decreasing order of their weights. Further, let \( e'_1 e'_2 \ldots e'_i \ldots e'_m \) be the edges of an optimal solution in decreasing order - (Is \( m = n \)?). Since the weight of the greedy solution is not optimal, there must a \( j \leq n \) such that \( e_j < e'_j \). Let \( A = \{ e \in S | w(e) \geq w(e'_j) \} \). The subset \( \{ e_1, e_2 \ldots e_{j-1} \} \) is maximal with respect to \( A \) (Why?). All the elements in \( \{ e'_1, e'_2 \ldots e'_j \} \) form an independent subset of \( A \) that has greater cardinality. This contradicts Property 3. \( \square \)
Example 3.3 Half Matching Problem Given a directed graph with non-negative edge weights, find out the maximum weighted subset of edges such that the in-degree of any node is at most 1.

The problem defines a subset system where $S$ is the set of edges and $M$ is the family of all subsets of edges such that no two incoming edges share a vertex. Let us verify Property 2 by considering two subsets $S_p$ and $S_{p+1}$ with $p$ and $p + 1$ edges respectively. $S_{p+1}$ must have at least $p + 1$ distinct vertices incident on the $p + 1$ incoming edges and there must be at least one vertex is not part of $S_p$’s vertex set incident to $S_p$’s incoming edges. Clearly, we can add this edge to $S_p$ without affecting independence.

Example 3.4 Weighted Bipartite Matching

Consider a simple graph with a zig-zag. There are two maximal independent sets (set of edges that do not share an end-point), one with cardinality 2 and the other having only 1 edge. There Property 3 is violated.

3.2.1 Maximal Spanning Tree

Let us try to verify the exchange property. Let $F_1$ and $F_2$ be two forests such that $F_2$ has one edge less than $F_1$. We want to show that for some $e \in F_1 - F_2$, $F_2 \cup \{e\}$ is a forest. There are two cases

Case 1 $F_1$ has a vertex that is not present in $F_2$, i.e. one of the end-points of an edge, say $e' \in F_1$ is not present in $F_2$. Then $e'$ cannot induce a cycle in $F_2$.

Case 2 The set of vertices in $F_1$ is the same. The set of end-points of $F_1$ may be a proper subset of $F_2$. Even then we can restrict our arguments to the end-points of $F_1$ as $F_2$. Since there are more edges in $F_1$, the number of connected components in $F_2$ is more than $F_1$. Recall that if $v_F, e_F, c_F$ represent the the number of vertices, edges and connected components in a forest $F$ then

$$v_F - c_F = e_F$$

(Starting from a tree removal of an edge increases components by 1)

We shall show that there is an edge $(u', v')$ in $F_1$ such that $u'$ and $v'$ are in different connected components of $F_2$ and therefore $(u', v') \cup F_2$ cannot contain a cycle. If you imagine coloring the vertices of $F_1$ according to the components in $F_2$, at least one component (tree) will have vertices with more than one color from pigeon hole principle. Therefore there will be at least one edge that will have its end-points colored differently. Trace the path starting from one color vertex to a vertex with a different color - we will cross an edge with different colors on its end-point. This edge can be added to $F_2$ that connects two components.
Exercise 3.4 The matroid theory is about maximizing the total weight of a subset. How would you extend it to finding minimum weighted subset - for example Minimal Spanning Trees?

3.2.2 A Scheduling Problem

We are given a set of jobs $J_1, J_2, \ldots, J_n$, their corresponding deadlines $d_i$ for completion and the corresponding penalties $p_i$ if a job completes after deadlines. The jobs have unit processing time on a single available machine. We want to find out the minimize the total penalty incurred by the jobs that are not completed before their deadlines. Stated otherwise, we want to maximize the penalty of the jobs that get completed before their deadlines.

A set $A$ of jobs is independent if there exists a schedule to complete all jobs in $A$ without incurring any penalty. We will try to verify Property 2. Let $A, B$ be two independent sets of jobs with $|B| > |A|$. We would like to show that for some job $J \in B$, $\{J\} \cup A$ is independent. Let $|A| = m < n = |B|$. Start with any feasible schedules for $A$ and $B$ and compress them, i.e. remove any idle time between the jobs by transforming the schedule where there is no gap between the finish time of a job and start time of the next. This shifting to left does not affect independence.

Let us denote the (ordered) jobs in $A$ by $A_1, A_2, \ldots, A_m$ and the times for scheduling jobs in $A$ be $d_1, d_2, \ldots, d_m$ respectively. Likewise, let the jobs in $B$ be $B_1, B_2, \ldots, B_n$ and their scheduling times $d'_1, d'_2, \ldots, d'_n$.

If $B_n \notin A$, then we can add $B_n$ to $A$ and schedule it as the last job. If $B_n = A_j$, then move $A_j$ to the same position as $B_n$ (this doesn’t violate the deadline) creating a gap at the $j$-th position in $A$. We can now shift-to-left the jobs in $A - A_j$ and now by ignoring the jobs $B_n = A_j$, we have one less job in $A$ and $B$. We can renumber the jobs and are in a similar position as before. By applying this strategy inductively, either we succeed in adding a job from $B$ to $A$ without conflict or we are in a situation where $A$ is empty and $B$ is not so that we can now add without conflict.

3.3 Efficient data structures for MST algorithms

The greedy algorithm described in the previous section is known as Kruskal’s algorithm that was discovered much before the matroid theory was developed. In the usual implementation of Kruskal’s algorithm, the edges are sorted in increasing order of their weights.
Algorithm Kruskal_MST

**input** Graph $G = (V, E)$ and a weight function on edges.

**output** A minimum Spanning Tree of $G$

Let $e_1, e_2 \ldots e_m$ be the elements of $E$ in increasing order of weights. Initialize $T = \emptyset$.

For $i = 1$ to $m$ do

In the $i$-th stage

If $T \cup \{e_i\}$ doesn’t contain a cycle, then $T \leftarrow T \cup \{e_i\}$

Output $T$.

The key to an efficient implementation is the *cycle test*, i.e., how do we quickly determine if adding an edge induces a cycle in $T$. We can view Kruskal’s algorithm as a process that starts with a forest of singleton vertices and gradually connects the graph by adding edges and growing the trees. The algorithm adds edges that connect distinct trees (connected components) - an edge whose endpoints are within the same tree may not be added since it induces cycles. Therefore we can maintain a data structure that supports the following operations

**Find** For a vertex, find out which connected component it belongs to.

**Union** Combine two connected components.

For obvious reasons, such a data structure is called a union-find structure and can be seen in a more general context where we have a family of subsets and for any given element we can find the subset it belongs to and we can merge two subsets into one. The subsets are assumed to be disjoint.

### 3.3.1 A simple data structure for union-find

Let us try to use arrays to represent the sets, viz., in an array $A$, let $A(i)$ contain the label of vertex $i$. The labels are also in the range $1, 2 \ldots n$. Initially all the labels are distinct. For each set (label), we also have pointers to all its elements, i.e., the indices of the array that belong to the set.

**Find** Is really simple - for vertex $i$ report $A(i)$. This takes $O(1)$ time.

**Union** To do $\text{union}(S_j, S_k)$, we change the labels of all elements pointed to by $j$ and link them with $k$. Thus after union, we have labelled all the elements in the union as $k$. The time for this operation is proportional to the number of
elements in set \( j \). For obvious reasons, we would change labels of the smaller subset.

Although the time for a single union operation can be quite large, in the context of MST, we will analyze a sequence of union operations - there are at most \( n - 1 \) union operations in Kruskal’s algorithm. Consider a fixed element \( x \in S \). The key to the analysis lies in the answer to the following question.

How many times can the label of \( x \) change?

Every time there is a label change the size of the set containing \( x \) increases by a factor of two (Why?). Since the size of a set is \( \leq n \), this implies that the maximum number of label changes is \( \log n \). Kruskal’s algorithm involves \(|E|\) finds and at most \(|V| - 1\) unions; from the previous discussion this can be done in \( O(m + n \log n) \) steps using the array data-structure.

3.3.2 A faster scheme

The previous data structure gives optimal performance for \( m \in \Omega(n \log n) \) so theoretically we want to design better schemes for graphs with fewer edges. For this we will explore faster schemes for union-find.

Instead of arrays, let us use trees\(^3\) to represent subsets. Initially all trees are singleton nodes. The root of each tree is associated with a label (of the subset) and a rank which denotes the maximum depth of any leaf node. To perform Find \( x \), we traverse the tree starting from the node \( x \) till we reach the root and report its label. So the cost of a Find operation is the maximum depth of a node.

To perform Union \((T_1, T_2)\), we make the root of one tree the child of the root of the other tree. To minimize the depth of a tree, we attach the root of the smaller rank tree to the root of the larger rank tree. This strategy is known as the union by rank heuristic. The rank of a tree increases by one only when \( T_1 \) and \( T_2 \) have the same ranks. Once a root node becomes a child of another node, the rank doesn’t change (by convention). The union operation takes \( O(1) \) steps.

**Exercise 3.5** Prove that

(i) The number of nodes in tree of rank \( r \) is at least \( 2^r \) if we use the union by rank heuristic.
(ii) The maximum depth of a tree (or equivalently the value of any rank) is at most \( \log n \).
(iii) There are at most \( \frac{n}{2^r} \) nodes of rank \( r \).
(iv) The ranks of nodes along any path from a node to the root are increasing monotonically.

\(^3\) this tree should not be confused with the MST that we are trying to construct.
So we are in a situation where Find takes \(O(\log n)\) and Union operation takes \(O(1)\). Seemingly, we haven’t gained anything so let us use the following heuristic.

**Path compression**

When we do a Find\((x)\) operation, let \(x_0 = \text{root of } x, x_1, x_2 \ldots x\) be the sequence of nodes visited. Then we make the subtrees rooted at \(x_i\) (minus the subtree rooted at \(x_{i+1}\)) the children of the root node. Clearly, the motivation is to bring more nodes closer to the root node, so that the time for the Find operation decreases. And Path compression does not increase the asymptotic cost of the current Find operation (it is factor of two).

While it is intuitively clear that it should give us an advantage, we have to rigorously analyze if it indeed leads to any asymptotic improvement.

### 3.3.3 The slowest growing function?

Let us look at a very rapidly growing function, namely the tower of two. The tower \(i\) looks like

\[
2^{2^{\ldots^2}}
\]

which can be defined more formally as a function

\[
B(i) = \begin{cases} 
2^1 & i = 0 \\
2^2 & i = 1 \\
2^{B(i-1)} & \text{otherwise for } i \geq 2
\end{cases}
\]

Let

\[
\log^{(i)} n = \begin{cases} 
n & i = 0 \\
\log(\log^{(i-1)} n) & \text{for } i \geq 1
\end{cases}
\]

The inverse of \(B(i)\) is defined as

\[
\log* n = \min\{i \geq 0 | \log^{(i)} n \leq 1\}
\]

In other words,

\[
\log* 2^{2^{\ldots^2}} = n + 1
\]

We will use the function \(B()\) and \(\log*()\) to analyze the effect of path compression. We will say that two integers \(x\) and \(y\) are in the same block if \(\log* x = \log* y\).

Although \(\log*\) appears to slow down anything we can imagine, (for example \(\log* 2^{65536} \leq 5\)), there is a closely related family of function called the inverse Ackerman function that is even slower!
Ackerman’s function is defined as

\[
A(1, j) = 2^j \quad \text{for } j \geq 1 \\
A(i, 1) = A(i - 1, 2) \quad \text{for } i \geq 2 \\
A(i, j) = A(i - 1, A(i, j - 1)) \quad \text{for } i, j \geq 2
\]

Note that \(A(2, j)\) is similar to \(B(j)\) defined earlier. The inverse-Ackerman function is given by

\[
\alpha(m, n) = \min \{i \geq 1 \mid A(i, \lfloor \frac{m}{n} \rfloor) > \log n\}
\]

To get a feel for how slowly it grows, verify that

\[
\alpha(n, n) = 4 \quad \text{for } n = \sqrt[2]{2^{16}}
\]

### 3.3.4 Putting things together

Clearly the cost of Find holds key to the analysis of this Union Find data structure. Since the rank is less than \(\log n\), we already have an upperbound of \((\log n)\) for any individual Find operation. We will adopt the following strategy for counting the cost of Find operations. We will associate a hypothetical counter with each node that we will increment whenever it is visited by some Find operation. Finally, by summing the counts of all the nodes, we can bound the cost of all the find operations. We will refer to the incremental cost of Find operation as a charge.

We further distinguish between two kinds of charges

**Block charge** If the block number of the parent node is strictly greater then a node incurs a block charge. Clearly the maximum number of block charges for a single Find operation is \(O(\log^* n)\)

**Path charge** Any charge incurred by a Find operation that is not a block charge.

From our previous observation, we will focus on counting the path charges.

**Observation 3.1** Once the rank of a node and its parent are in different blocks, they continue to be in different blocks, i.e. once a node incurs block charges, it will never incur any more path charges.

The parent of a node may change because of path compression preceded by one or more union operations, but the new parent will have a rank higher than the previous parent. Consequently, a node in block \(j\) can incur path charges at most \(B(j) - B(j -
1) \leq B(j) \text{ times. Since the number of elements with rank } r \text{ is at most } \frac{n}{2^r}, \text{ the number of elements having ranks in block } i \text{ is }

\frac{n}{2^{B(i-1)+1}} + \frac{n}{2^{B(i-1)+2}} + \cdots + \frac{n}{2^{B(i)}} = n \left( \frac{1}{2^{B(i-1)+1}} + \frac{1}{2^{B(i-1)+2}} + \cdots \right) \leq 2n \frac{1}{2^{B(i-1)+1}} = \frac{n}{2^{B(i-1)}}

Therefore the total number of path charges for elements in block } i \text{ is at most } \frac{n}{2^{B(i-1)}} \cdot B(i) \text{ which is } O(n). \text{ For all the log}^* n \text{ blocks the cumulative path charges is } O(n \log^* n) \text{ to which we have to add } O(m \log^* n) \text{ block charges.}

Remark: For some technical reason, in a Find operation, the child of the root node always incurs a block charge (Why?)

3.3.5 Path compression only

If we only use path-compression, without the union-by-rank heuristic, we cannot bound the rank of a node by log } n. \text{ Since the union-by-rank heuristic does not change the asymptotic bound of the union operation, it is essentially to gain a better understanding of role of the path compression.}

Note that the ranks of the nodes from any node the root still increases monotonically - without the union-by-rank heuristic, the rank of node can increase by more than one (in fact arbitrarily) after a union operation. Let us denote parent of a node } x \text{ as } p(x), \text{ parent of parent of } x \text{ as } p^2(x) \text{ and likewise. Let us define the level of a node } x \text{ by } \ell(x) \text{ as an integer } i \text{ such that } 2^{i-1} \leq \text{rank}(p(x)) - \text{rank}(x) \leq 2^i. \text{ Therefore } \ell(x) \leq \log n.

We account for the cost of a find(x) operation by charging a cost one to all the nodes in the path from } x \text{ to the root. The only exception is that for any level } i, \ 1 \leq i \leq \log n, \text{ the last node (in the path to the root) in level } i \text{ is not charged. Instead the cost is charged to the find operation. Clearly the number of charges to the find operation is } O(\log n). \text{ For any other node } y, \text{ note that the } \ell(y) \text{ increases at least by one from the monotonicity of ranks and the fact that it is not the last node of its level. Therefore over the course of all the union-find operations, a node can get charged at most } \log n \text{ times resulting in a total cost of } O(m \log n) \text{ for all the find operations.}

Exercise 3.6 Can you construct an example to show that the above analysis is tight?