# Parallel Addition And Subtraction In $O(1)$ Time And $O(n * lg(n))$ Space: Short Paper

Matthew Zupan

26/11/2024

# Contents

# 1 Introduction

For this short paper we will specify the meaning of certain notations for enhanced readability:
$\vee$ means OR, $\wedge$ means AND, $\oplus$ means XOR, $\neg$ means NOT, COR means 'concurrent OR', that it is $x_1 \vee x_2 \vee x_3 \vee ... \vee x_k$ simultaneously. COR has the same meaning as the symbol $\bigvee$, however, COR represents a physical logic gate unit. Lastly, $\bigwedge$ is the symbol for $x_1 \wedge x_2 \wedge x_3 \wedge ... \wedge x_k$, which is the symbol to AND together multiple items.

Parallel binary addition is achieved in $O(1)$ time and $O(n*lg(n))$ space utilising both the standard logical AND, OR and NOT gates as well as COR gates. The use of the COR gate model makes this equivalent to the complexity class AC given that we can represent unbounded fan-in for OR as COR and unbounded fan-in for AND as $\neg\text{COR}(\neg x_1, \neg x_2, ..., \neg x_n))$. Thus we are limited in this paper to the constraints of the AC complexity class and present an improvement over a previous result of time $O(1)$ and space $O(n^3)$. Here we allow for addition of integers $A$ and $B$ of size $n = 2^k$, where $k \in \mathbb{Z}, k > 0$ for a simpler analysis and use left to right indexing from 1 to $n$ inclusive. The result of addition $C$ uses left to right indexing from 0 to $n$ inclusive.

The addition of two integers $A$ and $B$ requires that for each digit at index $i$, we add together $A_i$ and $B_i$ into result $C_i$ along with an input carry $c_{i+1}$ which has been the result of addition from digits in $A$ and $B$ at index $i + 1$, illustrated in Figure 2.1. It is sufficient to break down the process of addition of two numbers $A$ and $B$ into three separate components. The first component is the computation of result $R = A_i \oplus B_i$ for $1 \leq i \leq n$. The second requirement is to determine for carries $C$, whether a given index $C_i$ for for $0 \leq i \leq n$ will receive a carry that has been propagated down from further along. Lastly we compute the output $O = R_i \oplus C_i$ for $0 \leq i \leq n$, where $R_0$ is set to 0 by default. Computation of $R$ and $O$ are trivial, thus we turn our attention to the issue of efficiently computing $C$.

First of all we will define the label $P$ to be a propagator for a given index $i$ that passes on an incoming carry of 1 to the left at $i - 1$. This occurs in the case where $(A_i, B_i) = (0, 1)$ or $(A_i, B_i) = (1, 0)$. This is because the result $R_i = (A_i \oplus B_i)$ is equal to 1 under this condition and therefore any incoming carry would result in $1 + 1 = 0$ carry 1. To avoid confusion of notation, the statement "index $i$ is of type $P$" will be expressed as $i :: P$.

The next label we define is $G$, referred to as a generator which generates a carry at some index $i$ if $(A_i, B_i) = (1, 1)$. This trivially occurs due to the fact that $1 + 1 = 0$ carry 1. The statement "index $i$ is of type $G$" will be expressed as $i :: G$.

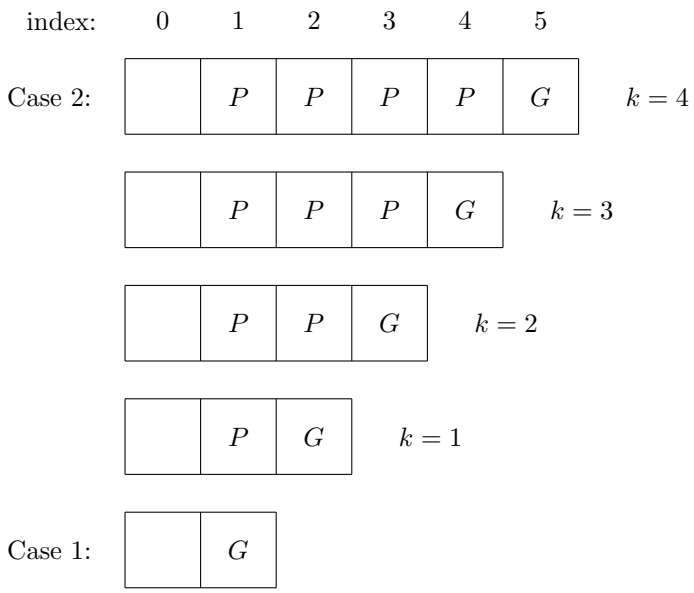
With the preceeding labels defined, we may now observe that a given element $C_i$ in $C$ can only receive a carry ($C_i = 1$) in the following two cases ($C_i = 0$ otherwise):

Case 1:
- Index $i + 1$ is a generator, $c_1 = i + 1 :: G$.

Case 2:
- An index $k$ greater than $i+1$ is a generator and all indices $j$ between $i$ and $k$ are propagators, $c_2 = \bigvee_{k=i+2}^{n}[(k :: G) \wedge (\bigwedge_{i+1}^{k-1}[j :: P])]$.

We are required to compute $C_i =$ Case 1 $\vee$ Case 2, as we will receive no carry if both evaluate to 0, and we will receive a carry if either evaluates to 1. The previous two cases are visually demonstrated in the following, Figure 1.1:

Figure 1.1: Case 1 and 2, Conditions Where Index $i = 0$ Receives Carry.




The difficulty in computing the carries in space less than $O(n^c)$, where $c$ is a real number greater than 1, arises in Case 2. This is due to the nature of repeating work by checking for propagators within the intervals $i + k$. By dividing into smaller blocks of intervals of size $m$, we effectively require $O(m^2)$ work per block. In the next section, we will see how we can maintain this narrative, yet reduce the space to $O(n * lg(n))$ with a neat trick.

# 2 Computing Carries

## 2.1 Tree Of Propagators ($T$)

We begin by setting up a register $P'$ corresponding to propagators ranging from indices $1 \leq i \leq n$. For each $P'_i$ in parallel, we compute $A_i \oplus B_i$, such that $P'_i = 1$ if $i :: P$ and 0 otherwise. For later use, we also set up the register $G'$ corresponding to generators ranging from indices $1 \leq i \leq n$. For each $G'_i$ in parallel, we compute $A_i \wedge B_i$, such that $G'_i = 1$ if $i :: G$ and 0 otherwise.

Next, we utilise a complete binary tree structure $T$ with $lg(n) + 1$ layers, indexed in $T$ with label $m$, where $0 \leq m \leq lg(n)$ to store intermediary information. We will define the root layer to be $T_{m=lg(n)}$ and the leaf layer to be $T_{m=0}$. Each layer thus has $n/2^m$ nodes, indexed in $T_m$ with label $q$, where $1 \leq q \leq (n/2^m)$. Each node has exactly one left and right child (except for the leaf layer which have no children). We can now properly index each node in T as $T_{(m,q)}$, and with a hefty abuse of notation, each node $T_{(m,q)}$ will store a single value $p_{(m,q)} = (P^{\forall} \rightarrow \{0, 1\})$, where "$P^{\forall}$" is read as "every leaf node reachable as a descendant from the current node must be of type $P$". The notation "$\rightarrow \{0, 1\}$" is read as "maps to the set {False, True}". Note that for the leaf nodes which have no children, they will store the same informtion in the same form, however only for their own singular value.

The index $q$ for each leaf node $T(0, q)$ directly corresponds to index $i$ in registers $P'$, and the index $q$ for nodes $T(m, q)$ in layers $1 \leq m \leq lg(n)$ directly correspond to the coverage of the range of indices $2^m(q - 1) + 1 \leq i \leq 2^m q$ in $P'$.

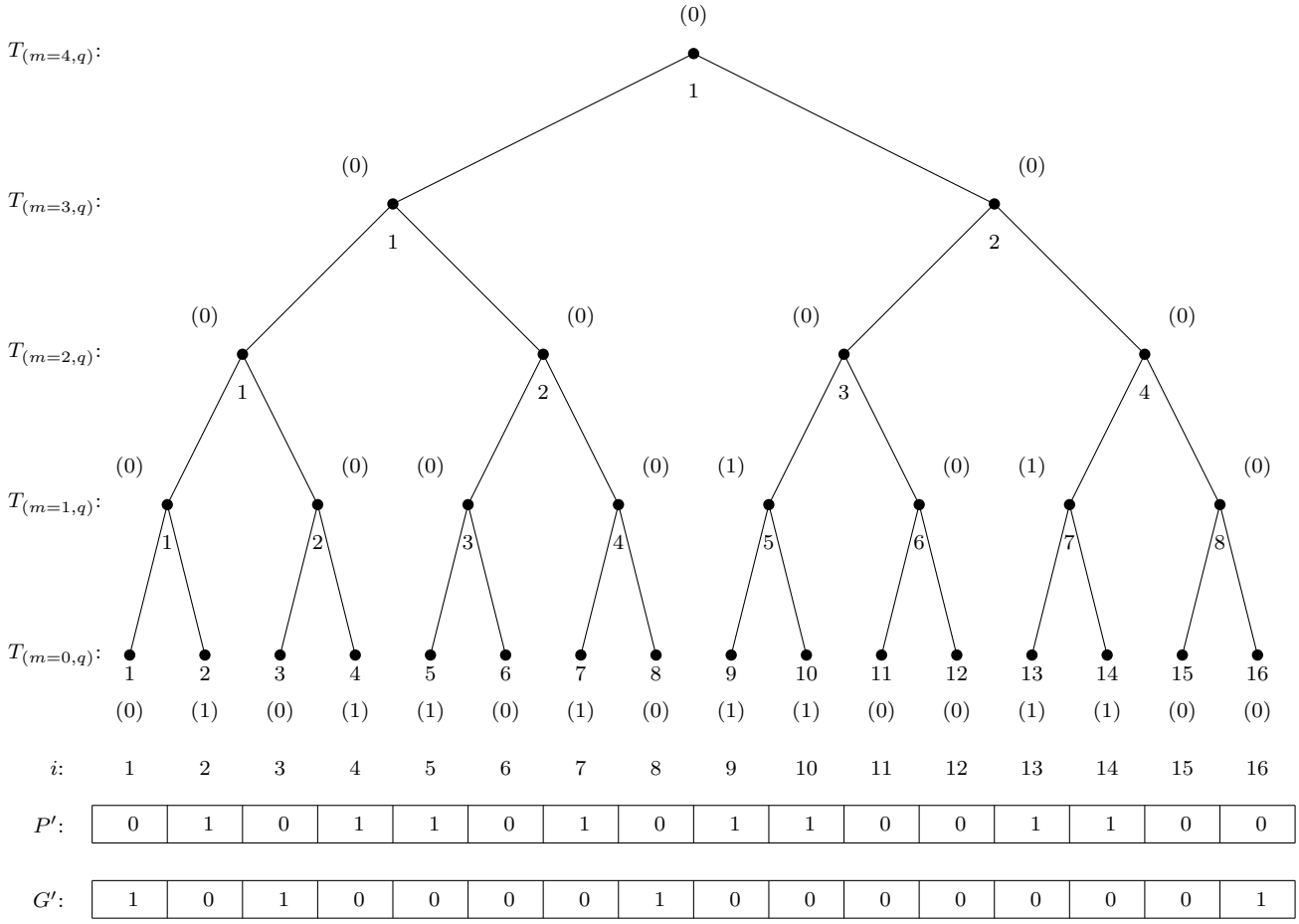
We compute the singleton $p_{(0,q)}$ for each $q$ in parallel trivially by taking $P^{\forall} = P'_{i=q}$. For the singleton $p_{(m>0,q)}$ for each $T_{(m>0,q)}$ in parallel, we compute $P^{\forall}$ as:

$P^{\forall} = \bigwedge_{i=2^m(q-1)+1}^{2^m q} P'_i$, which can be done by applying a $\neg COR$ gate to $P'$, with each element of $P'$ negated.

Summarily, we have $p_{(m=0,q)} = (P'_{i=q})$ and
$p_{(m>0,q)} = (\neg COR(\neg P'_i)$, where $2^m(q-1)+1 \leq i \leq 2^m q$, which takes $O(nlg(n))$ space as we visit each element of $P'$ once for each layer $m$ in $T$.

Figure 2.1.1 illustrates an example of a tree and the corresponding propagator singletons formed where $n = 16$, $A = 1011\ 0011\ 1100\ 1001$ and $B = 1110\ 1001\ 0000\ 0101$. These values will be used for all subsequent examples in this paper:

Figure 2.1.1: $T$, The Tree of Propagators ($P^{\forall}$).




## 2.2 Tree Of Generators and Propagators ($T^*$)

Our next requirement is to form another tree $T^*$ similar to the previous, where each node $T^*_{(m,q)}$ also stores the value $g_{(m,q)} \to \{0, 1\}$, labelled to represent the truth value of whether any leaves within the subtree $T'$ rooted from $T_{(m,q)}$ generate and propagate a carry to the node directly to the left of its first leaf (to index $2^m(q-1)$), according to case 1 and 2 in Figure 1.1.

Firstly, given that each node $T_{(m,q)}$ in $T$ is the root of its' own subtree, we will define the left sub-root $L_{(m,q')}$ to be any node in $T$ such that it is itself

a left-child within $T$ and contains its' own subtree $T'$, where $q' = 2q_{m+1} - 1$ (using the indexing $q$ from the layer above).

Next we require another definition. Each node $L_{(m,q')}$ has a set of $k$-right-left relatives, which are all nodes reachable as only left children of the right child of the parent of $L_{(m,q')}$. In this definition, $k$ is not a variable, but represents the difference in layer $\Delta m$ between nodes. Note that for leaf nodes $L_{(m=0,q')}$, there are only 0-right-left relatives which are direct siblings, all other nodes $L_{(m>1,q')}$ have $(k > 0)$-right-left relatives.

We formulate the connections for all left sub-roots to their $k$-right-left relatives as follows:

- Leaf nodes which are left sub-roots $L_{(m=0,q')}$ are connected to a set consisting of a singular $k$-right-left relative $\{T_{(m=0,q'+1)}\}$.

- For each left sub-root $L_{(m>0,q')}$ in $T$ and for each layer $m' \geq 0$ under $m$. $L_{(m,q')}$ is connected to each of its' $k$-right-left relatives in the set $\{T_{(m',q)} | (2q' - 1) * 2^{m-m'} + 1, 0 \leq m' < m\}$.

We have now formed a set of branched paths from each left root $L_{(m',q')}$ within each subtree $T'_{(m,q)}$, to each left leaf reachable as a descendent of $T'$. We also require an extra branch that extends each path from $L$ to reach all corresponding right leaves as well, with the left leaf sub-root connection $L_{(m=0,q')}$ to $T_{(m=0,q'+1)}$.

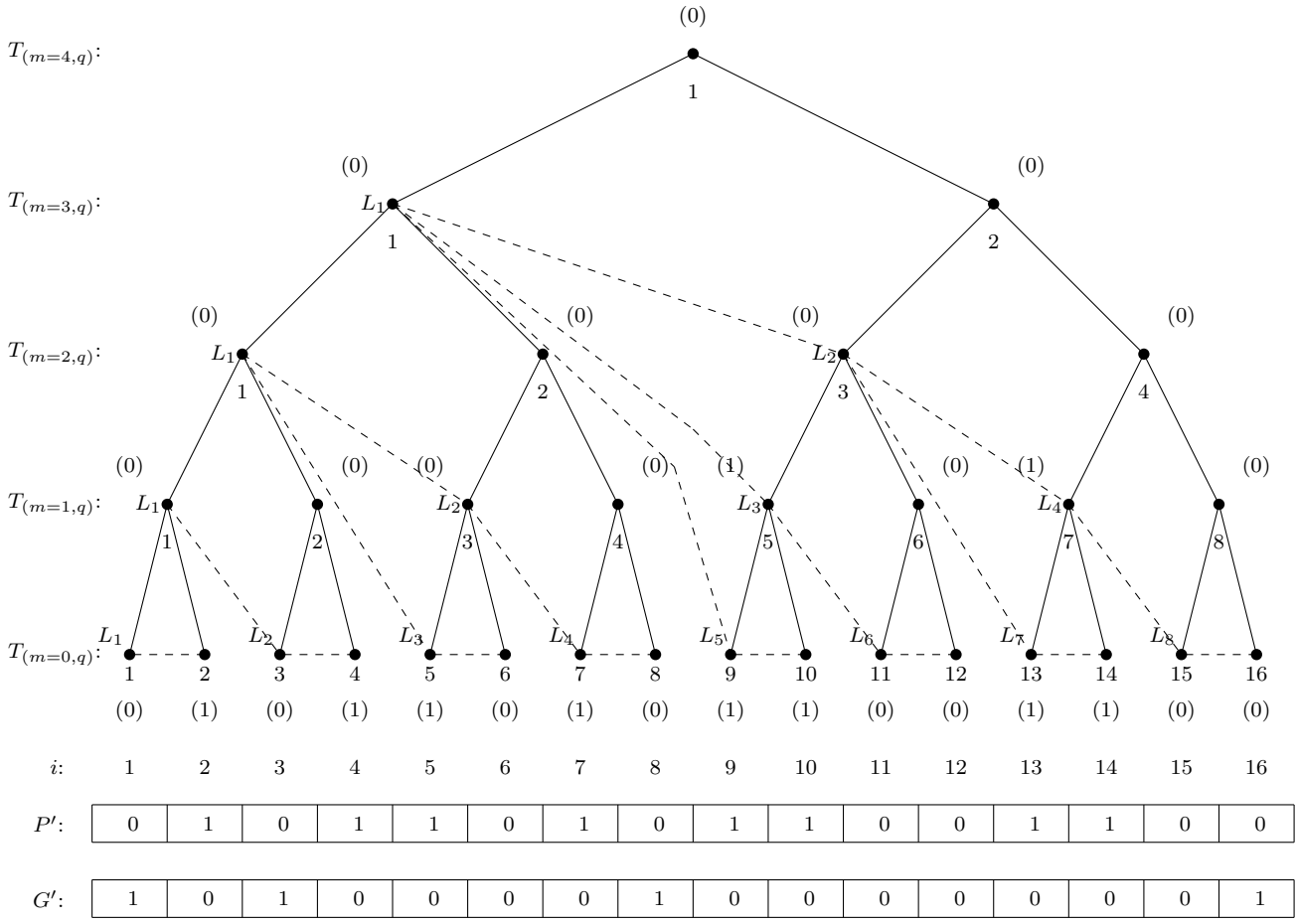
The outlined terminology and notation above will become more apparent when read in context with Figure 2.2.1, however, for now we will make a set of observations on the properties of left roots and their branches.

1. The length of the longest path is $lg(n)$ given that it descends from layer $m = lg(n) - 1$ to $m = 0$. Thus all other paths within $T$ are less than $lg(n)$ and can be bounded by length $O(lg(n))$.

2. The number of branched paths $b$ from the traversal of any given $L_{(m,q')}$ to its' connected leaves is given by $b_{(m,q')} = 2^m$. This is just the number of leaves reachable from as descendents from the right child of the parent of $L_{(m,q')}$.

3. For each subtree $T'_{(m,q)}$, each layer $m'$ in $T'$ has $2^{m-m'-1}$ left roots, and by observation 2, each $L_{(m',q')}$ has $2^{m'}$ branches. Therefore the sum of unique branches $b'$ contained within the subtree $T'$ is $b'_{(m,q)} = \Sigma_{m'=0}^{m-1}(2^{m-m'-1} * 2^{m'})$. Thus $b'_{(m,q)} = m * 2^{m-1}$.

Figure 2.2.1: $k$-right-left Relatives And Branched Paths In $T$.




In order to simplify analysis for the purpose of this short paper, we will avoid specifying an algorithm to evaluate our branched paths $L^*$ of $k$-right-left relatives from each $L$ to their corresponding leaves. However, given these set of paths, we must compute $g_{(m,q)}$ as outlined by cases 1 and 2 of Figure 1.1 over each path simultaneously in parallel. It is the fact that we have established $L^*$ that we can shortcut the process of brute forcing case 2 by accessing the singleton $p$ for each node on the path. This has the same effect of brute forcing testing the span of leaves in $T'$ as in Figure 1.1 but now with a log number of items to test, as the set $L^*$ for a given leaf $T_{(0,q_0)}$ forms a cover over $T'_{(0,2^m(q-1)+1)}$ to $T_{(0,q_0-1)}$. Now we compute case 1 and 2 as follows:

Case 1: Test the first leaf in $T'$ is of type $G$ by extracting the corresponding index in $G'$: $c_1(T'_{(m,q)}) = G'_{2^m(q-1)+1}$.

Case 2: We can formulate the test for case 2 of each $T'$ individually by considering all leaf starting positions together simulataneously, extracting $p_{(m,q')}$ from each $L_{(m',q')}$ in each path set $L^*(T_{(m,q)}, T'_{(0,q_0)})$ Note that to extract the coordinates of $p_{(m,q)}$ in $T'$ from each $L_{(m',q')}$, we can apply the transform $m = m'$ and $q = 2q' - 1$:

$$c_2(T'_{(m,q)}) = \bigvee_{q_2=2^m(q-1)+2}^{2^m q}[(G'_{q_2}) \wedge (\bigwedge_{l=1}^{|L^*(T'_{(m,q)},T'_{(0,q_0)})|} L^*(T'_{(m,q)}, T'_{(0,q_2)})_{(l,p)})].$$

Proceeding the computation of cases 1 and 2, we make an identical copy $T^*$ of the structure of $T$ to store both the previous values $p_{(m,q)}$ along with our values $g_{(m,q)} = c_1 \vee c_2$ into a tuple $g^*_{(m,q)} = (p_{(m,q)}, g_{(m,q)})$ for each $T'_{(m,q)}$. For the subtree $T'_{(3,2)}$ using Figure 2.1.1, an explicit description of the calculation will be provided below, with the results for $T^*$ illustrated in Figure 2.2.2:

Case 1: $c_1(T'_{(3,2)}) = G'_9$, corresponding to the leaf $T'_{(0,9)}$.

Case 2:

Constructions of all $L'$ in $T'_{(3,2)}$:

- $L^*(T_{(3,2)}, T'_{(0,10)}) = \{L_{(0,5)}\}$.
- $L^*(T_{(3,2)}, T'_{(0,11)}) = \{L_{(1,3)}\}$.
- $L^*(T_{(3,2)}, T'_{(0,12)}) = \{L_{(1,3)}, L_{(0,6)}\}$.
- $L^*(T_{(3,2)}, T'_{(0,13)}) = \{L_{(2,2)}\}$.
- $L^*(T_{(3,2)}, T'_{(0,14)}) = \{L_{(2,2)}, L_{(0,7)}\}$.
- $L^*(T_{(3,2)}, T'_{(0,15)}) = \{L_{(2,2)}, L_{(1,4)}\}$.
- $L^*(T_{(3,2)}, T'_{(0,16)}) = \{L_{(2,2)}, L_{(1,4)}, L_{(0,8)}\}$.

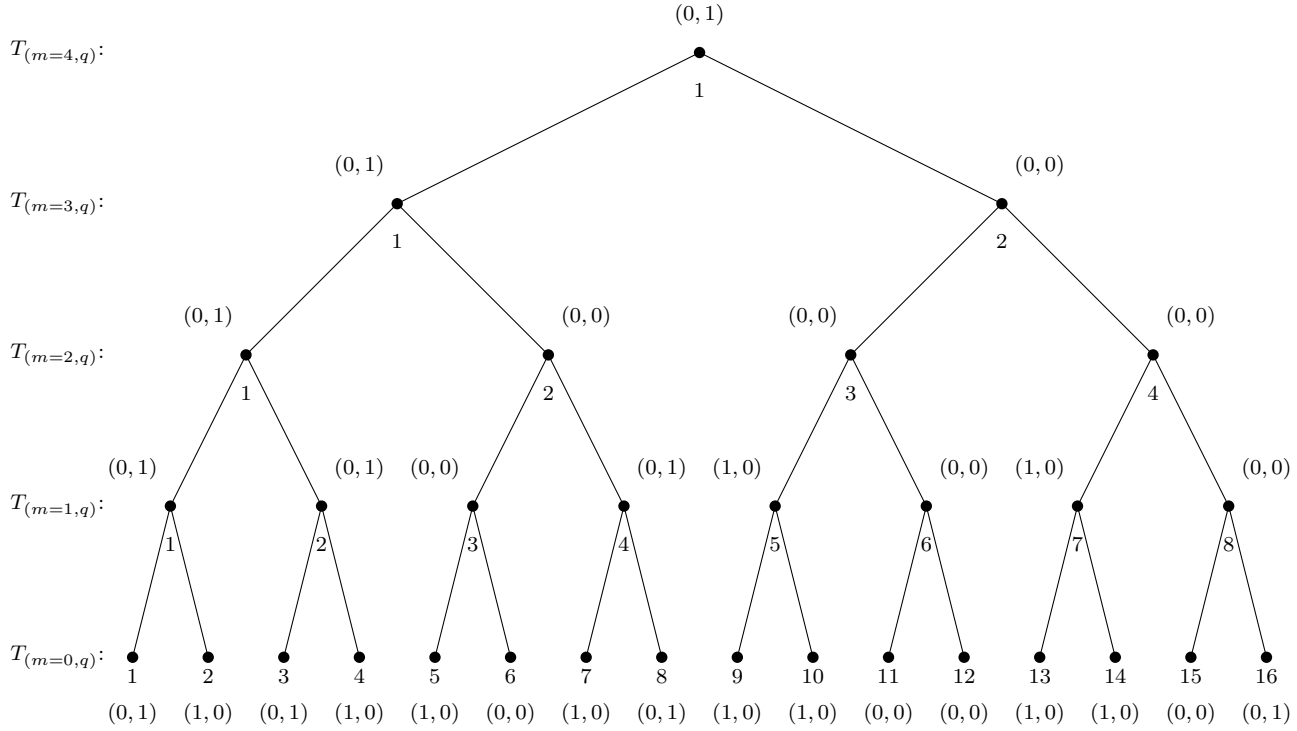
Use of all values $p$ from all nodes $L$ within the sets $L'$:

$$\begin{aligned} c_2(T'_{(3,2)}) = {} & [(G'_{10}) \wedge (L_{(0,5)})] \vee [(G'_{11}) \wedge (L_{(1,3)})] \vee [(G'_{12}) \wedge (L_{(1,3)} \wedge L_{(0,6)})] \vee \\ & [(G'_{13}) \wedge (L_{(2,2)})] \vee [(G'_{14}) \wedge (L_{(2,2)} \wedge L_{(0,7)})] \vee [(G'_{15}) \wedge (L_{(2,2)} \wedge L_{(1,4)})] \vee \\ & [(G'_{16}) \wedge (L_{(2,2)} \wedge L_{(1,4)} \wedge L_{(0,8)})] \end{aligned}$$

Next, using cases 1 and 2, compute: $g_{(3,2)} = c_1(T'_{(3,2)}) \vee c_2(T'_{(3,2)})$.

Finally, form the tuple $g^*_{(3,2)}$ to be stored in $T^*_{(3,2)}$ by combining the corresponding $p_{(m,q)}$ and $g_{(m,q)}$: $g^*_{(m,q)} = (p_{(m,q)}, g_{(m,q)})$.

Figure 2.2.2: $T^*$, The Tree Of Generators and Propagators ($g^*$).




From observation 3, we determine that $b'_{(lg(n),1)} = lg(n) * 2^{lg(n)-1}$ for the entire tree $T_{(lg(n),1)}$, and thus $b'_{(lg(n),1)} = \frac{1}{2}nlg(n)$, which is on the order of $O(nlg(n))$ branches. From observation 1, each branch is bounded by depth $O(lg(n))$, and therefore any computation utilising all branches must take $O(nlg(n)) * O(lg(n)) = O(nlg^2(n))$ space. This however is not good enough, as the target space complexity is $O(nlg(n))$, which leads us to employ yet another trick building upon the previous work.

## 2.3 A Revised Tree Of Generators and Propagators ($T^*$)

There are two issues with the previous tree, that being that the number of branches to compute is of the order $O(nlg(n))$, and that over each branch, we must perform $O(lg(n))$ brute force work. So it is necessary to attempt to reduce both the number of branches and their length such that the space of our brute force approach is bounded by the target complexity $O(nlg(n))$.

We make an observation that the number of branches in $T$ from layer $m = 2$ to $m = 1$ is less than the number of branches from layer $m = 1$ to $m = 0$. So it may be worthwhile to compute some portion of the lower section of the tree (layers 1 to $m$) and then use these computed results to compute the upper portion of the tree (layers $m + 1$ to $lg(n)$).

Our choice will be $m = \sqrt{lg(n)}$, which means we will be brute force computing $\sqrt{lg(n)}$ layers ($0 \leq m' \leq m - 1$). Separately, layer $m = 0$ itself requires only $O(n)$ work in total as we already know their singular values from registers $P'$ and $G'$. We will now focus on this lower portion of $T$.

From observation 3, the total number of branches for a subtree $T'$ at a layer $m$ is given by $b'_{(m,q)} = m * 2^{m-1}$. With our choice of $m$, we have $b'_{(\sqrt{lg(n)},q)} = \frac{1}{2}\sqrt{lg(n)} * 2^{\sqrt{lg(n)}}$.

At layer $m = \sqrt{lg(n)}$ in $T$, we have a total of $n/2^m$ nodes which are themselves subtrees containing their own branches, giving us a total of $\frac{1}{2}\sqrt{lg(n)} * 2^{\sqrt{lg(n)}} * n/2^{\sqrt{lg(n)}} = O(n * \sqrt{lg(n)})$ branches.

Lastly, given that each branch must then be bounded by $O(\sqrt{lg(n)})$ height, we achieve a space complexity for the lower portion of the tree of $O(n * \sqrt{lg(n)}) * O(\sqrt{lg(n)}) = O(nlg(n))$.

Now having computed the lower portion, we can use the top layer of the lower portion as the new 'leaves' of the upper tree. The upper tree now is a single tree which has height $lg(n) - \sqrt{lg(n)}$. The number of branches is then $lg(n) - \sqrt{lg(n)} * 2^{lg(n)-\sqrt{lg(n)}-1}$ using $m$ as the height of this upper tree. We calculate the space required in evaluating all branches simulataneously bounded by $O(lg(n))$ height as as $O(lg(n)) * [lg(n) - \sqrt{lg(n)}] * 2^{lg(n)-\sqrt{lg(n)}-1}$.

For simplicity, let $k = lg(n)$, then we have $k * [k - \sqrt{k}] * 2^{lg(n)-\sqrt{k}-1}$. Note that in the square bracket, the $\sqrt{k}$ term is asymptotically dominated by the $k$ term, so we will further simplify analysis such that the worst case now becomes: $k * k * n/(2 * 2^{\sqrt{k}})$.

The constant 2 is irrelevant to the asymptotic behaviour and we note that $2^{\sqrt{k}} > k$ for $n > 2^{16}$. We can trivially reduce this bound by having first computed the bottom $2 * \sqrt{lg(n)}$ layers for the bottom tree, leaving the upper tree with height $lg(n) - 2\sqrt{lg(n)}$ but that is besides the point. Asymptotically for some $n > c$, in our case $c = 2^{16}$, we can claim that $2^{\sqrt{k}} > k$. Thus we can lazily cancel one of the values $k$ from the numerator with $2 * 2^{\sqrt{k}}$ from the denominator,

leaving us with $k*n$. Substituting back in $k = lg(n)$, we have space $O(n*lg(n))$.

The generation of this tree $T^*$ overall occured in two parts, generation of the bottom tree with space $O(n * lg(n))$, followed storing and using these values to generate the top tree also with space $O(n*lg(n))$. Both components together are additive, leading to the asymptotic space complexity remaining at $O(n * lg(n))$ for $T^*$.

We will illustrate imperfectly here the general idea of both the bottom and top trees for clarity, labelled $T_b^*$ and $T_t^*$ respectively:

Figure 2.3.1: $T_b^*$, The Tree Of Generators and Propagators ($g^*$).

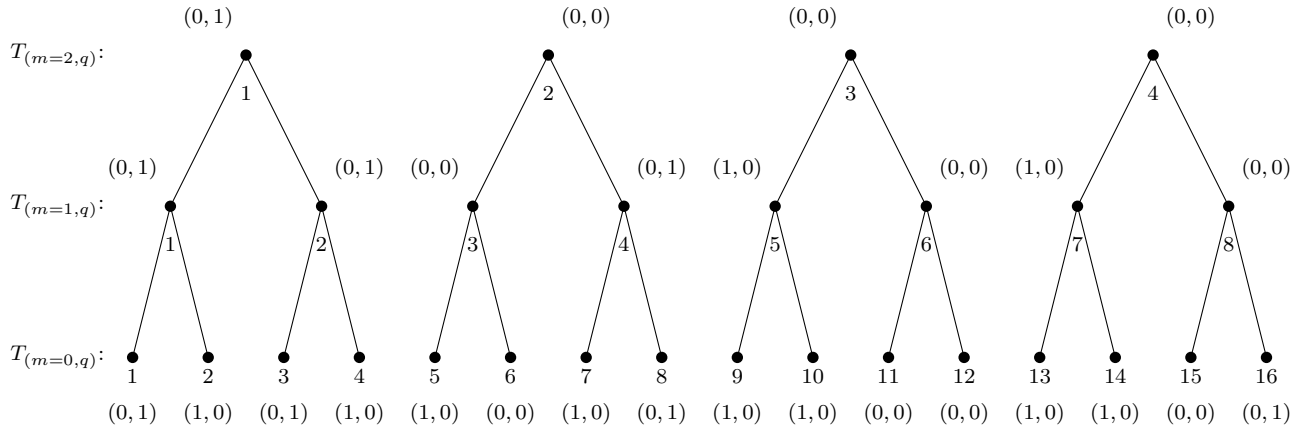
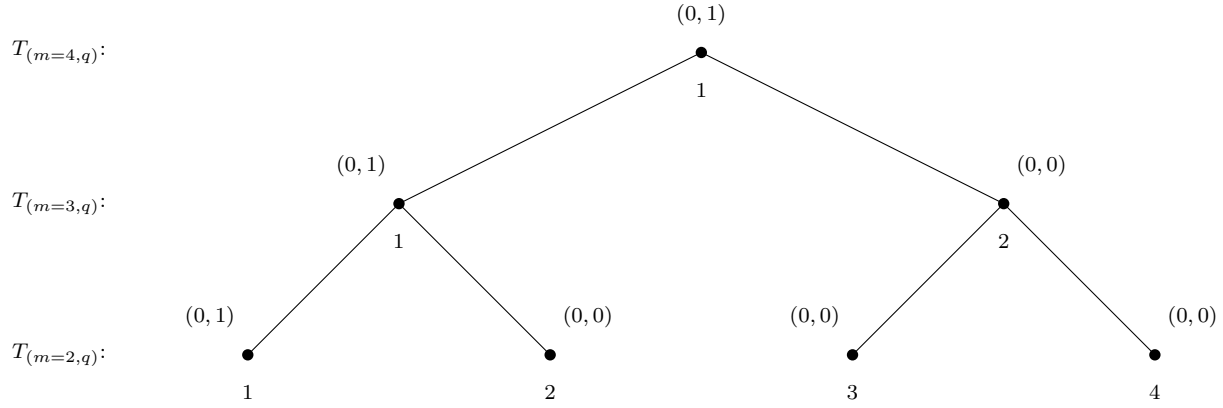


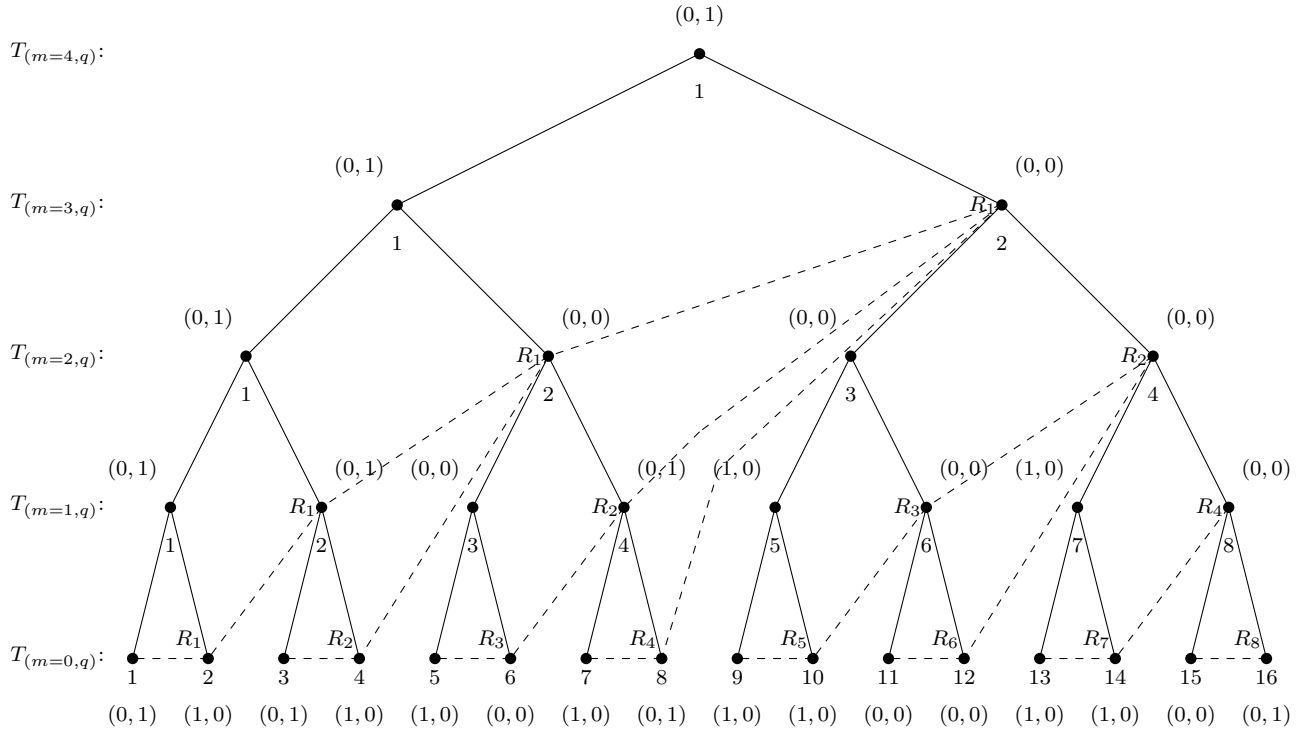

Figure 2.3.2: $T_t^*$, The Tree Of Generators and Propagators ($g^*$).



## 2.4 Using $T^*$ To Compute Carries

As in Section 2.2, we had defined the left roots $L$ and their corresponding $k$-right-left relatives, we shall lazily define in this section their mirror, the right roots $R$ and their corresponding $k$-left-right relatives which we will illustrate with a mirrored-branch tree $\bar{T}^*$ in the following diagram Figure 2.4.1:

Figure 2.4.1: $\bar{T}^*$, Mirrored-Branch Tree Of Generators and Propagators ($g^*$).



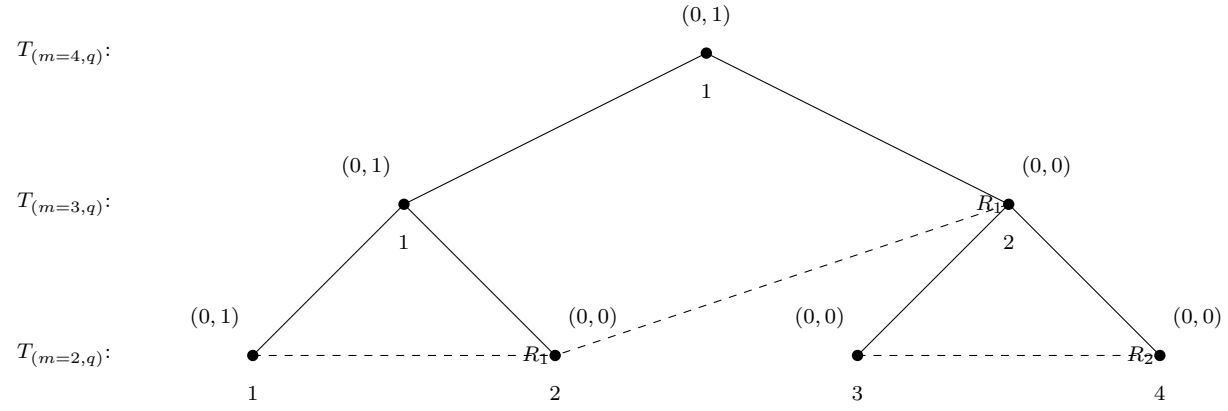

The next step in the process towards determining carries, is to cast back the carry information to the leaf nodes by performing Case 1 and 2 from Figure 1.1 along these mirrored branches. The difference being this time that the branches themselves must be traversed a quadratic number of times, as opposed to the leaf nodes previously being traversed a quadratic number of times in Section 2. The elements $P$ in this case are $p_{(m,q)}$, and the elements $G$ are $g_{(m,q)}$ from $g^*_{(m,q)} = (p_{(m,q)}, g_{(m,q)})$ in Section 2.2.

The details will be omitted as this is essentially the inverse process and follows the same narrative. However, we must be somewhat cautious and employ the trick from Section 2.3 by first evaluating the upper $lg(n) - 2 * \sqrt{lg(n)}$ layers, before using the intermediate results (the bottom layers of the top tree) to evaluate the bottom tree. The choice of multiplicative constant 2 is to reduce

our requirement $n > c$ from being impractically large.

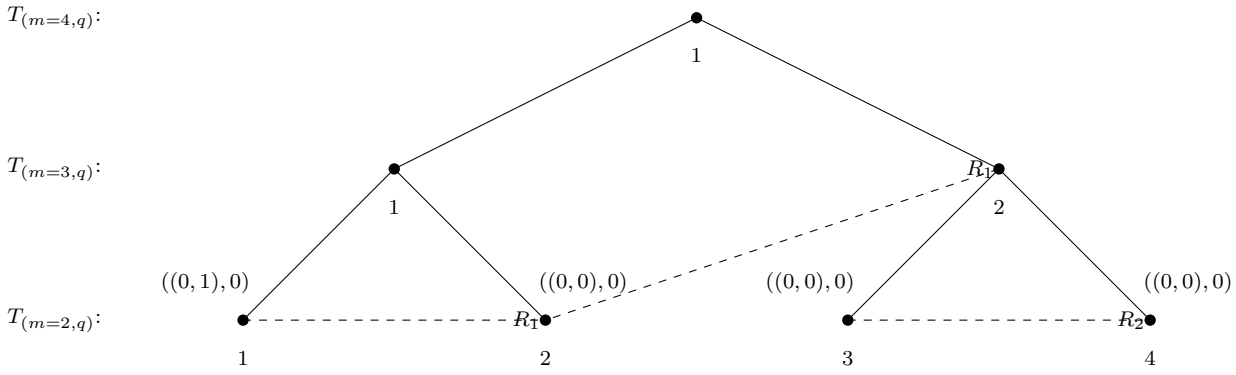
By taking Case 1 and 2 for only the upper layer $m = lg(n) - 2 * \sqrt{lg(n)}$, we generate information on whether these non-leaf nodes receive a carry or not. We will store this information in the nested tuple $c^* = (g^*, c)$, where $c \to \{0, 1\}$, and refers to our intermediate carry information stating that "the last leaf in the cover beneath this node in the bottom tree receives a carry". Firstly, due to the splitting of $\bar{T}^*$ into the top and bottom tree, we will need to treat $\bar{T}_t^*$ as a rescaled version of $\bar{T}^*$, with branches as illustrated as in Figure 2.4.2:

Figure 2.4.2: $\bar{T}_t^*$, Mirrored-Branch Tree Of Generators and Propagators ($g^*$).



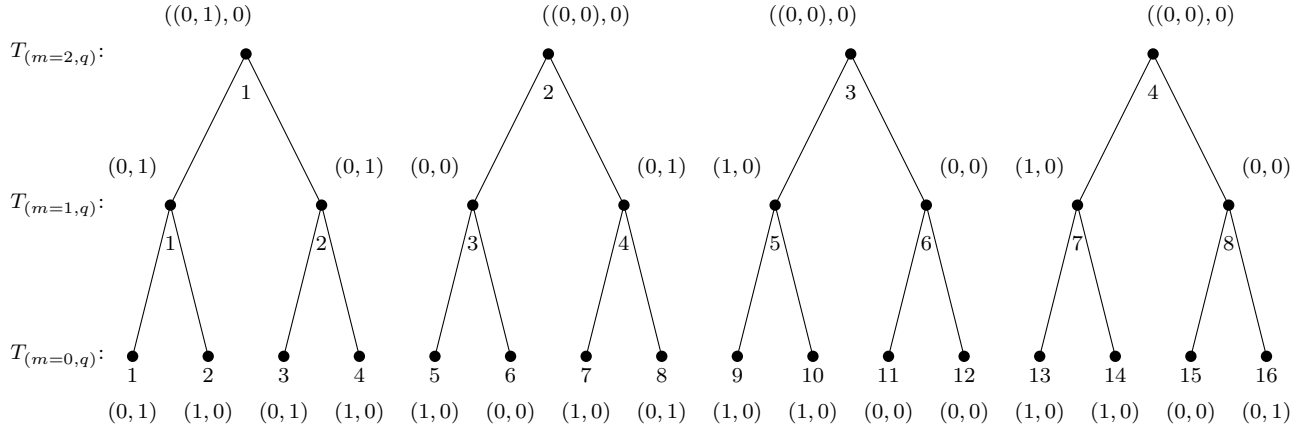

Via the previously specified computation, we form the tree $\bar{T}_t'$ with results $c^*$ in the leaves as shown in Figure 4.2.3 (the last leaf implicity receives a carry of 0):

Figure 2.4.3: $\bar{T}_t'$, Mirrored-Branch Tree Carry Information ($c^*$).



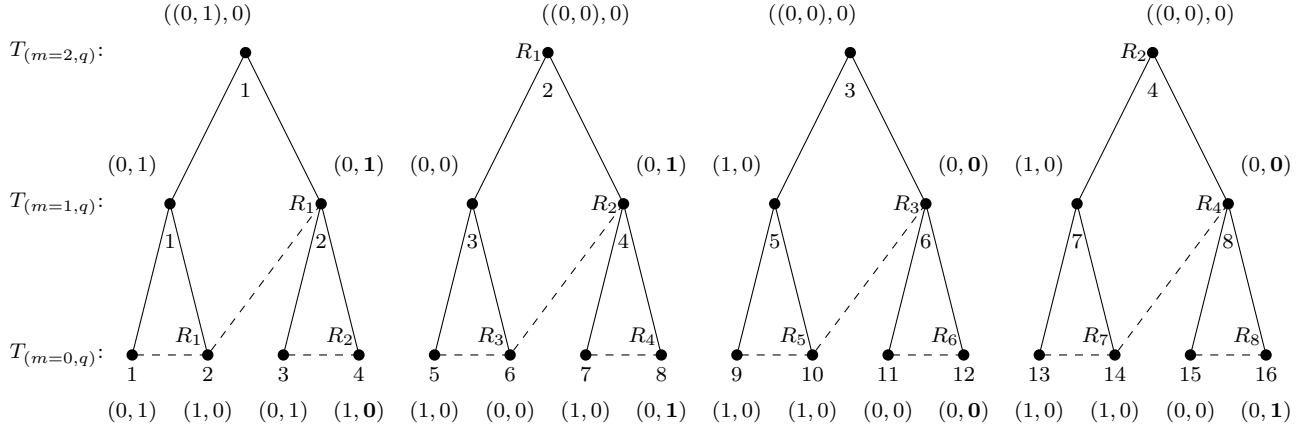
The leaves of $\bar{T}'_t$ form the roots of sub-trees in $\bar{T}^*_b$ containing the tuples $c^*$ respectively as shown in Figure 2.4.4:

Figure 2.4.4: $\bar{T}^*_b$, The Tree Of Roots ($c^*$).




Each directly right descendant from each respective root then updates its' value in parallel $g_{(m,q)}$ via the function $g_{(m,q)} = (c_{\text{ROOT}} \wedge p_{(m,q)}) \vee g_{(m,q)}$. Which has the effect of updating the 'generate and propagate' variable to now also include the potential that the last leaf has received a carry and the cover propagates it. This updated tree $\bar{T}'_b$ is illustrated in Figure 2.4.5 (branches re-inserted and only extending to the layer below the roots):

Figure 2.4.5: $\bar{T}'_b$, Mirrored-Branch Updated Tree ($c^*$).



All that is left to do, is to evaluate each leaf along its branch spanning right roots $R$ according to Case 1 and 2, which directly generates its carry information $c$ except for the last leaf in each subtree which receives a carry if $c_{\text{ROOT}} = 1$. These carries $c_q$ for each leaf are stored in the register $C$ and we must include the special case $c_0$, computed by extracting the value $g_{\text{ROOT}}$ of the leaf $(m = 0, 1)$. The result is shown below as the register $C$ in Figure 2.4.6:

Figure 2.4.6: Carry Register $C$.

| $i$: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $C$: | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

The output result $O_q = R_q * C_q$ is computed as outlined in the Introduction, with $C_0$ directly inserted into $O_0$, or via the function $C_0 \oplus 0$.

To conclude our analysis that the space complexity is truly $O(n * lg(n))$, we observe that the computation $\bar{T}'_t$ takes space $O(n * lg(n))$ via the following analysis:

The height of the tree is $lg(n) - 2 * \sqrt{lg(n)}$ and the number of branches can be approximated to equal the number of leaf nodes $2^{lg(n)-2*\sqrt{lg(n)}} = n/2^{2\sqrt{lg(n)}}$. Each branch requires a quadratic amount of space to compute Cases 1 and 2, therefore with branch length being bounded by the height of the tree, the branch space is $(lg(n) - 2 * \sqrt{lg(n)})^2$, which by largest term can be bounded by $O(lg^2(n))$.

The space taken by $\bar{T}'_t$ is then equal to the number of branches $*$ branch space, therefore space $= n/2^{2\sqrt{lg(n)}} * O(lg^2(n))$. Note that $2^{2\sqrt{lg(n)}} > lg(n)$ for $n > 1$, therefore we can cancel out the denominator $2^{2\sqrt{lg(n)}}$ with one of the terms $lg(n)$ to achieve a result for space bounded by $O(n * lgn)$.

For the bottom tree $\bar{T}'_b$, we have as many roots as the top tree has leaves, thus the number of roots is $n/2^{2\sqrt{lg(n)}}$. The number of branches in each rooted tree is approximately the number of leaves, being $2^{2*\sqrt{lg(n)}}$. We have branch length bounded by the height of each rooted tree, being $2 * \sqrt{lg(n)}$, and each requiring $(2 * \sqrt{lg(n)})^2 = 4 * lg(n)$ space for computation of cases 1 and 2.

The space taken by $\bar{T}'_b$ is then equal to the number of roots $*$ the number of branches $*$ branch space, therefore space $= (n/2^{2\sqrt{lg(n)}}) * 2^{2*\sqrt{lg(n)}} * 4 * lg(n)$. The terms $2^{2*\sqrt{lg(n)}}$ cancel out, and given the constant 4 is irrelevant to the asymptotic complexity, the result for space is bounded by $O(n * lg(n))$.

All these major components bounded by space $O(n*lg(n))$ in the worst case, are additive only over a constant number of time steps and thus the computation of $A + B$ occurs in time $O(1)$ and space $O(n * lg(n))$.

As indicated by the title of this paper, this is a short paper and lacks the rigid formalisation required to be at a standard of general publication. It is intended to illustrate how addition in $O(1)$ time and $O(n * lg(n))$ space is to be achieved without a solid implementation, which may or may not be provided at a later time. Sections 2.3 and 2.4 lack the rigour of the previous sections to highlight the key insight required, without being bogged down in the details, and Section 3 providing a series of example schematics omitted.