1. Since the median of an unsorted array can be found in linear time, we can find the median first, set it as the pivot and then do the quick sort with the median.

We can find median with worst-case linear time by choosing an element that divides array balancely.

```
Quick Sort (Arr, f, l)        → a_f, a_{f+1}, ..., a_l
if f ≥ l then return
 pivot = find-median (Arr)
 i = f, j = l
 while i < j
    while a_j ≥ pivot and i < j
        j = j-1
    a_i ↔ a_j
    while a_i ≤ pivot and i < j
        i = i+1
    a_i ↔ a_j
 end while
 Quick Sort (f, j-1)
 Quick Sort (j+1, l)
```

2. (a) $T(n) = 2T(\frac{n}{3}) + 6n$

$a = 2,\ b = 3,\ f(n) = 6n$

$n^{\log_b a} = n^{\log_3 2} \doteq n^{0.63},\ \varepsilon \approx 0.37$

$f(n) = \Omega(n) = \Omega(n^{\log_b a + \varepsilon})$

$2 \cdot f(\frac{n}{3}) = 2 \times 6 \times \frac{n}{3} = 4n \le cf(n),\ c = \frac{2}{3}$

$T(n) = \Theta(n)$

(b) $T(n) = 2T(\frac{n}{2}) + 6n$

$a = 2,\ b = 2,\ f(n) = 6n$

$n^{\log_2 2} = n$

$f(n) = 6n = \Theta(n) = \Theta(n^{\log_b a})$

$\Rightarrow T(n) = \Theta(n^{\log_b a} \lg n)$

$= \Theta(n \lg n)$

(c)

3. We divide the array into two subarrays of equal size, and we find the midpoint of the array A[low, ..., high], denoted as mid.

The two subarrays become A[low, ..., mid], and A[mid+1, ..., high]

We can find the maximum sub array of A[low, ..., mid], and A[mid+1, ..., high] recursively, since both subproblems are smaller instances of the original problem.

For any subarray crossing the midpoint, it can be seen as itself made of two subarrays A[i, ..., mid] and A[mid+1, ..., j] ($low \le i \le mid$ and $mid < j \le high$)

Therefore, we just need to find maximum subarray of the form A[i, ..., mid] and A[mid+1, ..., j] and combine them, this takes $\Theta(n)$.

```
Find-Max-SubArr (Arr, low, high)
  if low == high
      return Arr[low]
  else mid = ⌊(low + high)/2⌋
      l_sum = Find-Max-SubArr (Arr, low, mid)
      r_sum = Find-Max-SubArr (Arr, mid+1, high)
      cross_sum = Find-Max-Cross-SubArr (Arr, low, mid, high)

  return max(max (l_sum, r_sum), cross_sum)
```

Analyze: Assume that the original problem size is a power of 2.

$T(n)$: the running time of finding maximum subarray of n elements.

$T(1) = 1$

The recursive case ($n > 1$), each subproblem size: $\frac{n}{2} \Rightarrow T(\frac{n}{2}) \times 2$ (l_sum, r_sum)

Finding max cross subArr takes $\Theta(n)$

$\Rightarrow T(n) = 2T(\frac{n}{2}) + \Theta(n)$

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ 2T(\frac{n}{2}) + \Theta(n), & n > 1 \end{cases}$$

$\Rightarrow$ solve master method

$a = 2, b = 2, f(n) = \Theta(n)$

$\Rightarrow T(n) = \Theta(n \lg n)$

4. u, v : n bit binary numbers

We can divide each of the numbers into two halves with equal size ($n/2$)

Let the two halves be denoted u_L, u_R and V_L, V_R.

We can do the multiplication like

$$\begin{array}{r} u_L \quad u_R \\ \times \quad V_L \quad V_R \\ \hline u_L V_R \quad u_R V_R \\ +) \; u_L V_L \quad u_R V_L \\ \hline u_L V_L \quad \boxed{(u_L V_R + u_R V_L)} \quad u_R V_R \end{array} \Rightarrow \begin{array}{l} a = u_L V_L \\ b = u_R V_R \\ c = (u_L + u_R)(V_L + V_R) \end{array} \Rightarrow \text{化簡為 3次相乘}$$

$(u_L V_R + u_R V_L) \rightarrow c - a - b$

To multiply u and v, we can recursively multiply the pairs of $\frac{n}{2}$ size numbers

Time complexity : $O(n^{\log_2 3})$

$n = 2^k$, recursion stop if $n=1$, we will do 3^k digit multiplication

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

$\Rightarrow a=3, b=2 \quad n^{\log_2 3} = n^{1.58} \Rightarrow T(n) = O(n^{\log_2 3})$

$f(n) = O(n) = O(n^{1.58})$

$(\varepsilon \doteq 0.58)$

5. (a) False

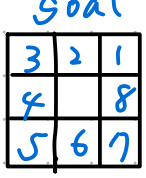
(b) True

For example, dynamic programming can provide better solution.

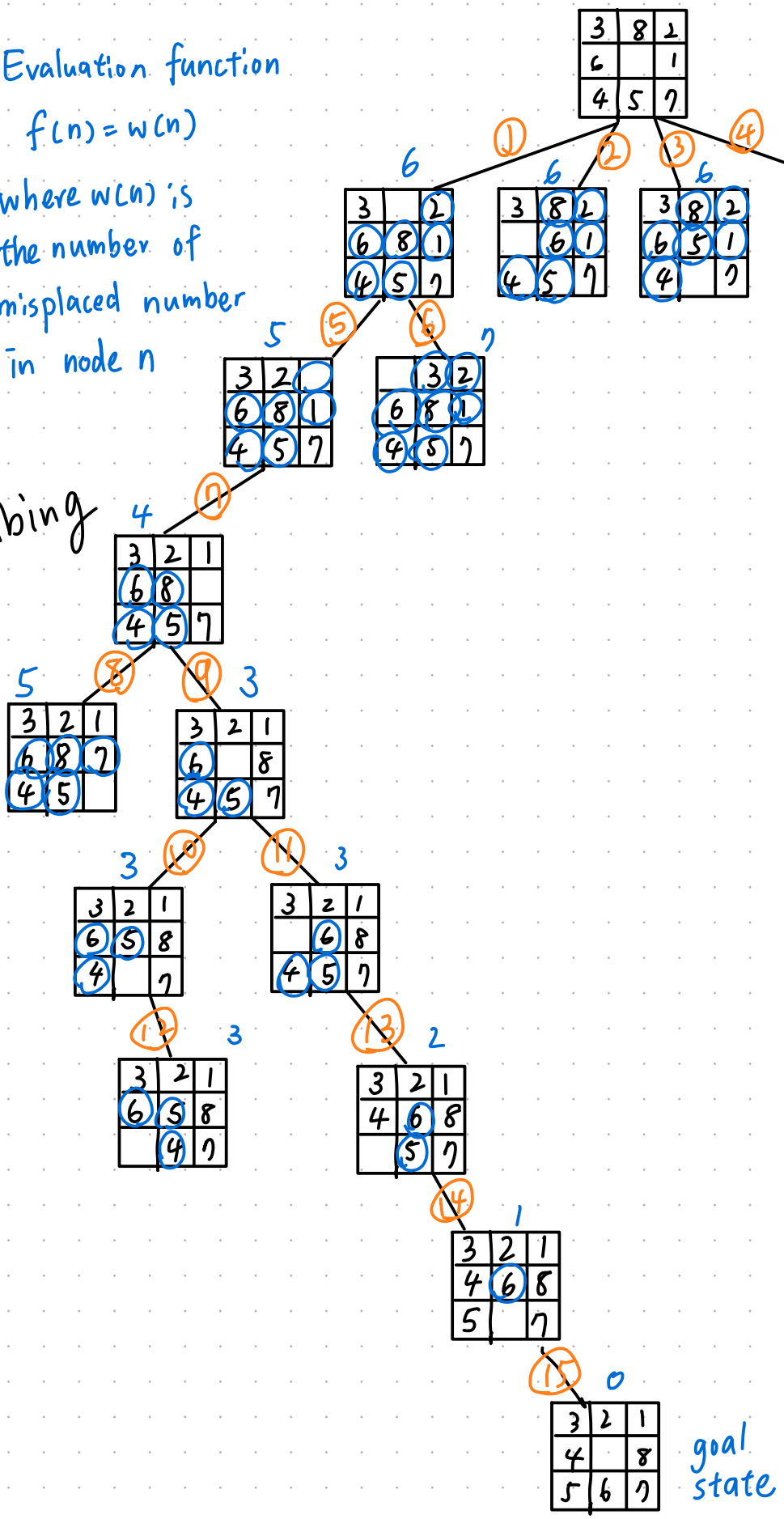
6. Evaluation function

$f(n) = w(n)$

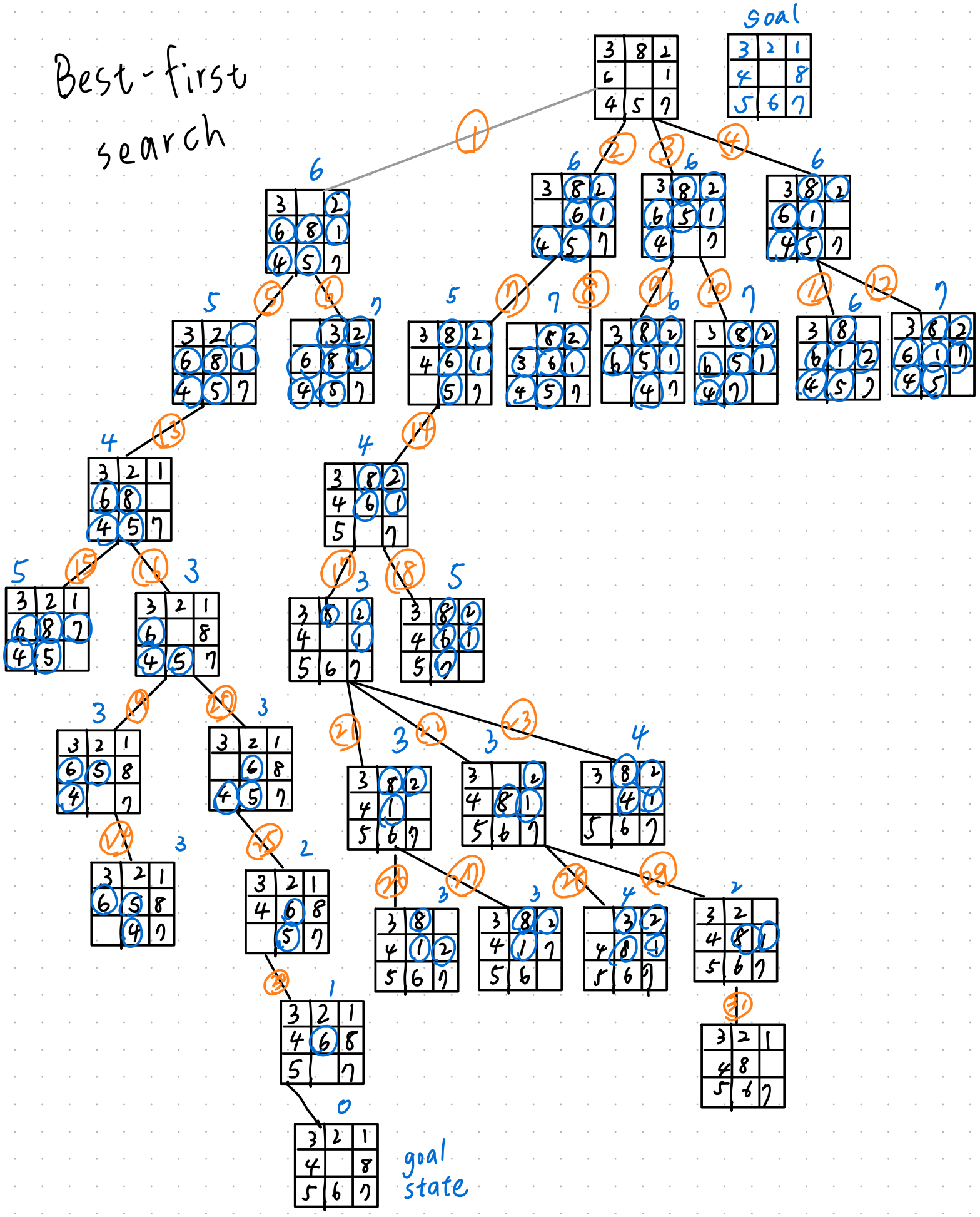
where $w(n)$ is the number of misplaced number in node n



Hill climbing

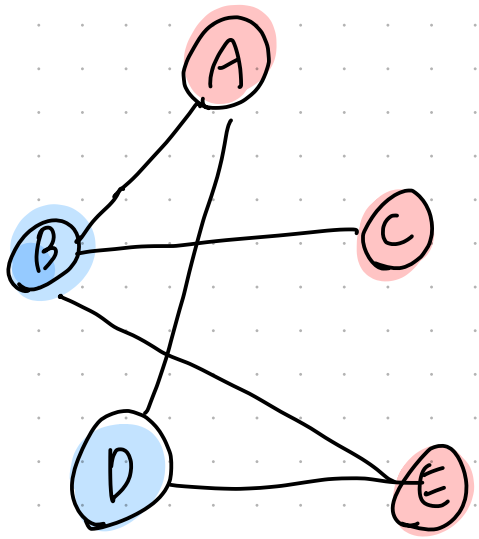


Best-first search



7. Solve this with BFS, set two color Red and Blue

1. Assign Red to the source vertex (A)
2. Color all neighbor with Blue (B, D)
3. Color all neighbor's neighbor Red



Yes, this graph is 2-colorable.

If we find a node that has the same color with its neighbor, the graph is not 2-colorable.

The time complexity is $O(n+e)$, n: # of nodes, e: # of edges.

Scanning for all neighbor node takes $O(e)$

And we need to scan through all nodes which takes $O(n)$

8. Because A* algorithm mentioned that under certain situations, an obtained feasible solution must be optimal ⇒ stop the algorithm.
It has the termination rule that if a node selected is the goal, then it represents an optimal solution, therefore stop the algorithm.

9.

i \ j	1	2	3	4	5
1	∞	6	62	35	13
2	57	∞	43	20	7
3	40	43	∞	9	22
4	6	50	43	∞	8
5	42	27	11	36	∞

subtract the highlighted number from each row.
And update LB.

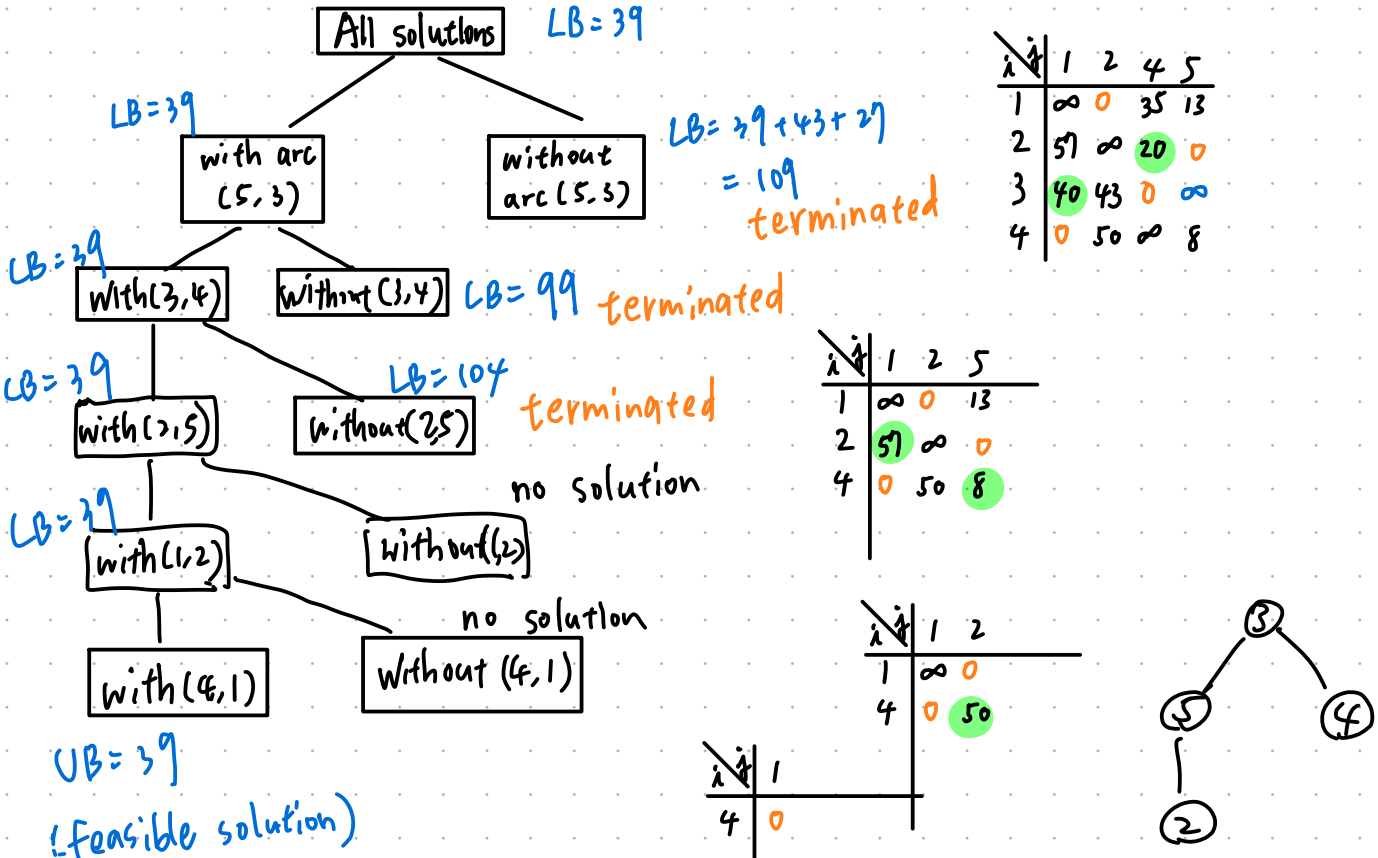
i \ j	1	2	3	4	5
1	∞	6	62	35	13
2	57	∞	43	20	7
3	40	43	∞	9	22
4	6	50	43	∞	8
5	42	27	11	36	∞

LB = 6+7+9+6+11 = 39

i \ j	1	2	3	4	5
1	∞	0	62	35	13
2	57	∞	43	20	0
3	40	43	∞	0	22
4	0	50	43	∞	8
5	42	27	0	36	∞

→ reduced cost matrix

⇒ find the arc that will cause largest increase of LB ⇒ arc (5,3)



i \ j	1	2	4	5
1	∞	0	35	13
2	57	∞	20	0
3	40	43	0	∞
4	0	50	∞	8

i \ j	1	2	5
1	∞	0	13
2	57	∞	0
4	0	50	8

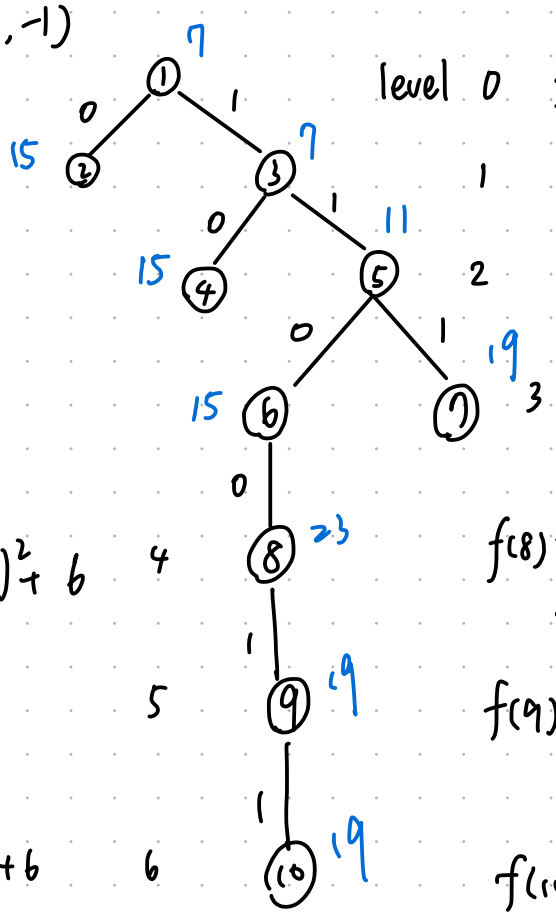
i \ j	1	2
1	∞	0
4	0	50

i \ j	1
4	0

10. 000000, 100110, 001011, 101101, 010101, 110011, 011110, 111000

The cost of branch from level $t-1$ to level $t = (r_t - (-1)^{c_t})^2$

$r = (-2, 1, 2, -2, -3, -1)$



level 0 $f(1) = g(1) + h(1)$

$= 0 + \sum_{i=1}^{6} (|r_i| - 1)^2$

$= 0 + (1+0+1+1+4+0)$

$= 7$

$f(2) = g(2) + h(2)$

$= (-2-(-1)^0)^2 + 6$

$= 15$

$f(3) = (-2-(-1)^1)^2 + h(3)$

$= 7$

$f(4) = (-2-(-1)^0)^2 + (1-(-1)^0)^2 + 6$

$= 15$

$f(5) = (-2-(-1)^1)^2 + (1-(-1)^1)^2 + 6$

$= 11$

$f(6) = (-2-(-1)^0)^2 + (1-(-1)^0)^2 + (2-(-1)^0)^2 + 5$

$= 15$

$f(7) = (-2-(-1)^1)^2 + (1-(-1)^1)^2 + (2-(-1)^1)^2 + 5$

$= 19$

$f(8) = 9+0+1+9+4$

$= 23$

$f(9) = 1+4+9+1+4+0$

$= 19$

$f(10) = 1+4+9+1+4+0+0$

$= 19$

closest code word is 110011.