complexity

```
for (i=2 ; i<n , i = i+2)
    Print (....)
```

$i=2 \rightarrow 1$ opt

$i<n \rightarrow \frac{n}{2} + 1$ opt

$i = i+2 \rightarrow \frac{n}{2}$ opt

print $\rightarrow \frac{n}{2}$ opt

$$t(n) = \sum_{i=1}^{\frac{n}{2}} 1 = t(n) = \frac{n}{2} \rightarrow O\left(\frac{n}{2}\right)$$

time complexity

```
for (i=1 , i≤n , i++)
    for (j=1 , j≤i , j++)
        Print (....)
```

$$t(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i$$

$$= \frac{n(n+1)}{2} = \frac{n^2+n}{2} = O(n^2)$$

$$t(n) = \frac{n^2+n}{2} \simeq O\left(\frac{n^2+n}{2}\right) = O(n^2+n) = O(n^2)$$

space complexity

```
for (i=1, i ≤ n , i = i*2)
    Print (....)
```

$$t(n) \sum_{i=1}^{\log_2 n} (1) = \log_2 n \rightarrow O(\log n)$$

```
for (i=2 , i≤n , i= i*i)
```

$2^{2^x} = n$

$2x = n$

$\log \log n$

$$\sum_{i=2} 1 = O(\log \log n)$$

i	# of steps	
2	2^1	2^{2^0}
4	2^2	2^{2^1}
16	2^4	2^{2^2}
256	2^8	2^{2^3}
⋮	2^{16}	2^{2^4}
n	2^k	2^{2^k}

$\log \log n \neq (\log n)^2$

$\log \log n = \log (\log n)$

```
for (i=1 , i<n , i++)
    for(j=1 , j≤n , j=j*2)
```

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{\log n} 1 = O(n \log n)$$

```
for (i=1; i<√n , i++ )
   for (j= 1 , j≤n/2 , j++)
          print (... )
          print(... )
```

$$t(n) = \sum_{i=1}^{\sqrt{n}} \sum_{j=1}^{n/2} 2 = \sqrt{n}\left(\frac{n}{2}\right)(2) = \sqrt{n}\,(n) = n^{1/2}(n^1) = n^{1/2+1} = n^{3/2}$$

$$= O(n^{3/2})$$

```
for ( i = 0,  i<(n-1) , i++ ) {
   for( j=i+1 , J<n , J++ ) {
      Print(.....)
   }
}
```

i	j	# of steps	
0	1, 2, 3, ..., n-1	n-1	n-i-1
1	2, 3, 4, ..., n-1	n-2	n-1-1
2	3, 4, 5, ..., n-1	n-3	n-2-1
⋮			
n-2	n-1	1	

$$t(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-i} 1$$

(upper limits: $n-1$ and $n-2+1$; lower limits: $i=0$ and $i=1$; inner limits: $j=i+1$ and $J=1$)

0, 1, 2, 3, 4, ... , n-2

remove 1 step

$$= \sum_{i=1}^{n-1} (n-i) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i$$

$$= (n-1)(n) - \frac{(n-1)n}{2}$$

$$= n^2 - n - \frac{n^2-n}{2}$$

$$= \frac{2n^2 - 2n - n^2 + n}{2} = \boxed{\frac{n^2-n}{2}} = O\left(\frac{n^2-n}{2}\right) = O(n^2)$$

n = 6

i	
0	1,2,3,4,5
1	2,3,4,5
2	3,4,5
3	4,5
4	5
5	
6	

# of steps

5
4
3
2
1

$5+4+3+2+1$

$= 15$ steps

time complexity $O(n^2) = \frac{6^2-6}{2} = \frac{36-6}{2} = \frac{30}{2} = 15$

space complexity $O(.1)$

repeat n times

```
for(i=1; i<n, i++)
  for(j=1, j≤i, j++)
    for(k=1, k≤j, k++)
      Print(....)
```

$$t(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=1}^{j} 1$$

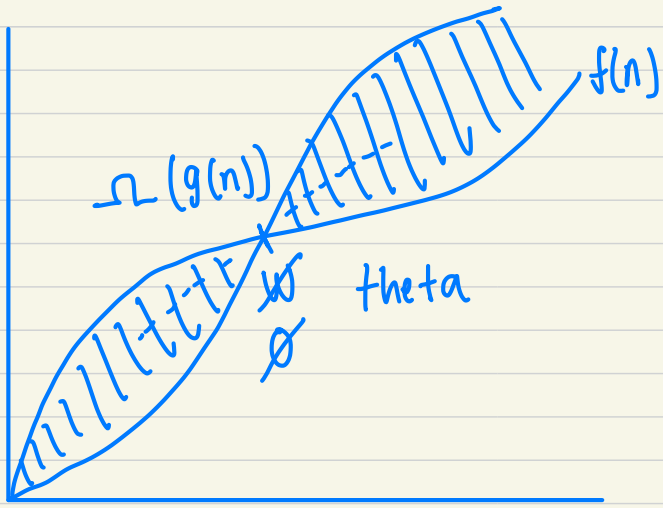
$$= \sum_{i=1}^{n} \left(\sum_{j=1}^{i} j \right)$$

$$= \sum_{i=1}^{n} \frac{i(i+1)}{2} = \frac{1}{2} \left(\sum_{i=1}^{n} i^2 + i \right)$$

$$= \frac{1}{2} \sum_{i=1}^{n} i^2 + \frac{1}{2} \sum_{i=1}^{n} i$$

$$= \frac{1}{2} \left(\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2} \right) = O(n^3)$$

(n^3 from the first term, n^2 from the second)



Θ = $f(n) = g(n)$

O = $f(n) \leq g(n)$

Ω = $f(n) \geq g(n)$

ω = $f(n) = g(n)$

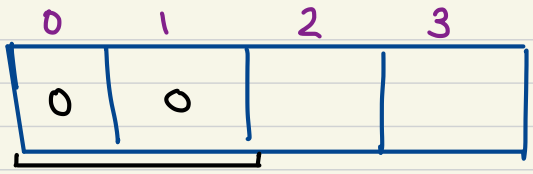
o = $f(n) < g(n)$

$t(n) = n^2 + n$

$\Omega(n^2)$

$\Omega(n)$

Average case complexity (sequential search)



the item is not in the array $\frac{1}{2}$

Worse case : $O(n)$

Best case : $O(1)$

Average case :

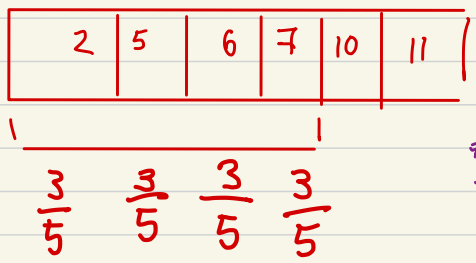
$$A(c) = \sum_{i=1}^{n} P_i \, i$$

$1+2+3+4+5+\ldots+n$

$$\frac{n(n+1)}{2}$$

$$P_i = P_{out} + P_{in} = 1$$

$$\sum_{i=1}^{n} 1$$

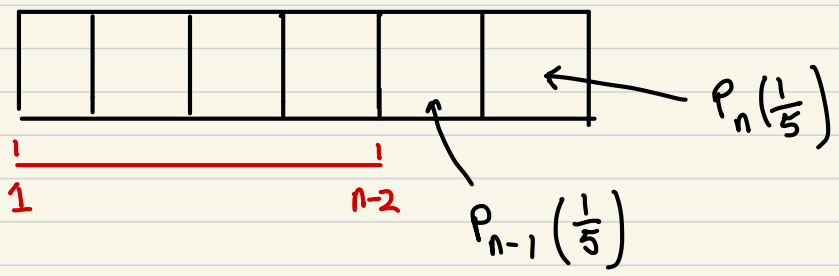


numbers of position with the same prob

$$P_{out} = \frac{1}{2} \cdot n$$

$$P_{in} = \frac{1}{2} \cdot \frac{1}{n}$$ ← not always n

could be $n-2$ as example



$$P_{out} = 0$$

$$P_{in} = \underbrace{P_n\left(\tfrac{1}{5}\right)}_{P_{out}} + \underbrace{P_{n-1}\left(\tfrac{1}{5}\right)}_{P_{out}} + \underbrace{P(1,\ldots n-2)\left(\tfrac{3}{5}\right)}_{P_{in}}$$

$$P_{out} + P_{in} = 1$$

$$1 = \frac{1}{5} + \frac{1}{5} + x$$

$$1 - \frac{1}{5} - \frac{1}{5} = x$$

$$= \frac{3}{5}$$

$$P_{in} = P_i \cdot \frac{1}{n}$$

$$P_{out} = P_i \cdot n$$

$$A(n) = P_{out} + P_{in} \sum_{i=1}^{n-2}$$

$$A(n) = 0 + \frac{n}{5} + \frac{n-1}{5} + \frac{3}{5(n-2)} \sum_{i=1}^{n-2}$$

$$= 0 + \frac{2n-1}{5} + \frac{3}{5(n-2)} * \frac{(n-2)(n-1)}{2} = \frac{2n-1}{5} + \frac{3(n-3)}{10}$$

$$= \frac{2(2n-1)+3n-3}{10} = \frac{4n-2+3n-3}{10} = \frac{7n-5}{10}$$

$n = 5$

$$\frac{7(n) - 5}{10} = \frac{7(5) - 5}{10} = \frac{30}{10} = 3$$

$P_n = \frac{1}{3}$

$P_{out} = 1/4$

$$P_{in} = \left(P_n(1/3)(3/4) + P_{n-1}(1/3)(3/4) + P(1 \ldots n-2) \right) = 3/4$$

$$\downarrow$$

$$P_n(\tfrac{1}{4}) + P_{n-1}(\tfrac{1}{4}) + P(1 \ldots n-2) \Big) = 3/4$$

$$A(n) = n/4 + n/4 + \frac{n-1}{4} + \frac{1}{4(n-2)} \cdot \sum_{i=1}^{n-2} i$$

$$= \frac{3n-1}{4} + \frac{1}{4(n-2)} * \frac{(n-2)(n-1)}{2}$$

$$\frac{3n-1}{4} + \frac{(n-1)}{8}$$

$$= \frac{6n-2+n-1}{8} = \frac{7n-3}{8}$$

CSC 510 Analysis of Algorithms Order - Asymptotic Bonding

Five types of notation

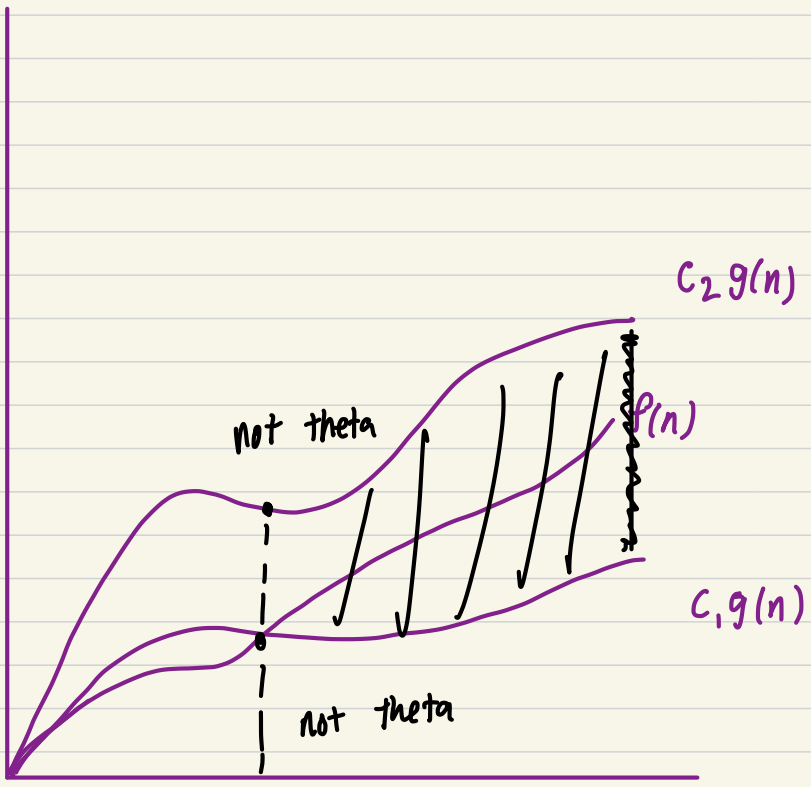
- Theta Notation (θ)
- Big O Notation (O)
- Small o notation (o)
- Omega notation (Ω)
- Small

Theta Notation (θ)

- Represents the upper and lower bounds of the running time in an algorithm.

Example

Prove that $\underbrace{4n+1}_{f(n)} = \underbrace{\theta(n)}_{g(n)}$



Big O notation Example

Prove that $\underbrace{n^2+3n+5}_{f(n)} = O(\underbrace{n^3}_{g(n)})$

$f(n) \leq g(n)$

$f(n) = n^2$

$g(n) = n^3$

Omega notation Example

Prove $\underbrace{n^2+100}_{f(n)} = \Omega(\underbrace{n}_{g(n)})$

$\Omega(n)$

This can be $\omega(n)$ cause of $f(n) > g(n)$

$f(n) = n^2$

$g(n) = n$

$f(n) > g(n)$

$\omega = f(n) > g(n)$

$o = f(n) < g(n)$

small o notation

Big Omega Notation

Prove $\underbrace{2n+1}_{f(n)} \neq \Omega(\underbrace{n^2}_{g(n)})$

$f(n) = n$

$g(n) = n^2$

ϵ : In discrete math is set

$O(g(n))$

$o(g(n))$

$f(n) < g(n)$

$$\frac{\log n\,(2^n)}{n!\; n^n\; 2^n\; \log n} + n! \in \Theta\,(2^{2^n} + n!)$$

Limit ratio review

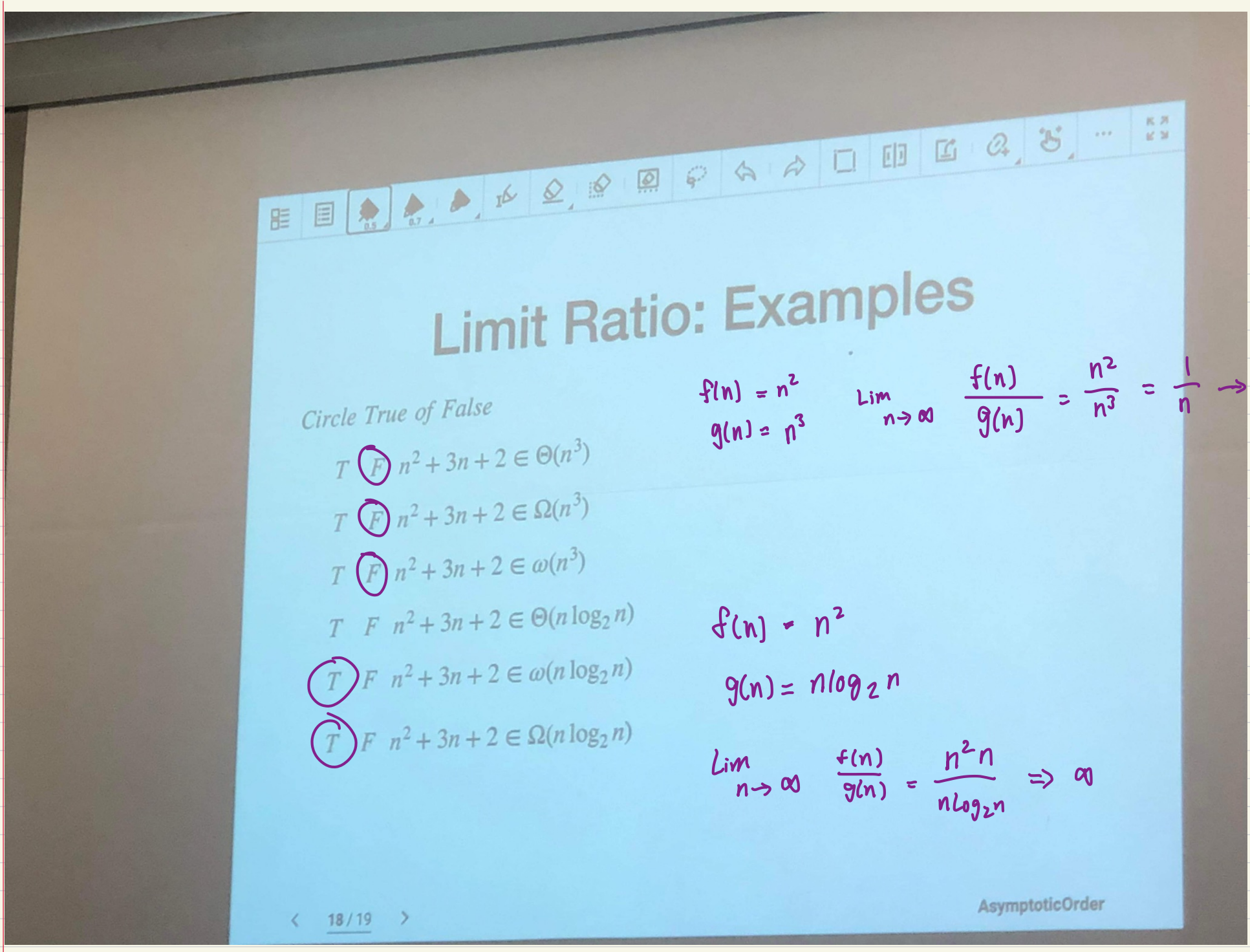
$$= \frac{2n^2 - 3n}{3n^2 + 3n} = \frac{n^2}{n^2} \to 1$$

$$= \frac{4n^2 + n + 1}{2n^3 - 5} = \frac{n^2}{n^3} = \frac{1}{n} \to \text{zero}$$

$$\lim_{n \to \infty} = \frac{3n^2 - 2n}{n + 5} = \frac{n^2}{n} \to \infty$$

Limit Ratio Notation Table

$O(g(n))$	$f(n) \leq g(n)$	$< \infty$
$o(g(n))$	$f(n) < g(n)$	zero
$\Omega(g(n))$	$f(n) \geq g(n)$	> 0
$\omega(g(n))$	$f(n) > g(n)$	∞
$\Theta(g(n))$	$f(n) = g(n)$	$\mathbb{R} > 0$ and $< \infty$



$f(n) = n^2$ $g(n) = n^3$ $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{n^2}{n^3} = \frac{1}{n}$ → Zero

$f(n) = n^2$

$g(n) = n \log_2 n$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{n^2 n}{n \log_2 n} \Rightarrow \infty$

(T) $3\log_2 n + 5n \in O(n \log_2 n)$

$f(n) = n$ $g(n) = n \log_2 n$

$\lim_{n\to\infty} \frac{n}{n \log n} = \frac{1}{n \log n}$ → zero

(T) $3^n \in \Omega(n^2 - n + 2)$

$f(n) = 3^n$

$g(n) = n^2$ ∞

$3n + 2 \log_2 n + 1 \in \theta \frac{(2^{\log_2 n})}{n}$

$2^{\log_2 n} = n^{\log_2 n} = n$

$T(0) = 1$
Base case

$T(n-k) = T(0)$

$n-k=0$

$n=k$

$T(n) = T(0) + (n-(n-1)) + (n-(n-2)) + \dots + (n-(n-n))$

$t(n) = t(0) + (1 + 2 + 3 + 4 + \dots + n)$

$1 + \sum_{i=1}^{n} i$

$t(n) = 1 + \frac{n(n+1)}{2} \longrightarrow 1 + \frac{n^2+n}{2} = \Theta$

function f(n)

If n == 1 return → T(0) = 1
Print(...) → 1
$f(\frac{n}{2})$ → $T(\frac{n}{2})$

}

T(0) = 1

Rule

$b^a = c$

$a = \log_b c$

function f(n/2) }
If .....
Print (.....)
$f(\frac{n}{4})$

$t(n) = t(\frac{n}{2}) + 1$

$t(\frac{n}{2}) = [T(\frac{n}{2^2}) + 1] + 1$

$\frac{n}{2} \leftarrow \frac{(\frac{n}{2})}{2} = \frac{n}{2} \cdot \frac{1}{2} = \frac{n}{4} = \frac{n}{2^2}$

$t(\frac{n}{2}) = T(\frac{n}{2^2}) + 2$

$t(\frac{n}{2^2}) = [T(\frac{n}{2^3}) + 1] + 2$

$T(\frac{n}{2^2}) = T(\frac{n}{2^3}) + 3$

⋮

After k steps

$t(n) = T(\frac{n}{2^k}) + k$

Base case

Solve for k

$T(\frac{n}{2^k}) = T(0)$ { $\frac{n}{2^k} = 0$

$n = 2^k(0)$

$n = 0$

$T(\frac{n}{2^k}) = T(1)$ { $\frac{n}{2^k} = 1$

$n = 2^k$

$\log_2 n = k$

→ $t(n) = 1 + \log_2 n = \Theta(\log_2 n)$

```
function f(n) {
    If  n==1   return  ->  T(1) = 1
    for (i=0, i<n , i++ ) {  ⟶ O(n)
            Print (----)
            Print (----)
    }
     f(n/2)  -> T(n/2)
     f(n/2)  -> T(n/2)
    }
```

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

$$\boxed{= 2\,T\left(\frac{n}{2}\right) + n}$$

$$t\left(\frac{n}{2}\right) = 2\left[2T\left(\frac{n}{2^2}\right) + \left(\frac{n}{2}\right)\right] + n$$

$$T\left(\frac{n}{2}\right) = \boxed{2^2}\,T\left(\frac{n}{2^2}\right) + \boxed{2n}$$

$$T\left(\frac{n}{2^2}\right) = 2^2\left[2T\left(\frac{n}{2^3}\right) + \left(\frac{n}{2^2}\right)\right] + 2n$$

$$t\left(\frac{n}{2^2}\right) = 2^3\,T\left(\frac{n}{2^3}\right) + 3n$$

$\vdots$

k steps

$k = \log_2 n$

$$T(n) = 2^k\,t\left(\frac{n}{2^k}\right) + kn$$

$$t(n) = 2^{(\log_2 n)}(1) + n\log_2 n$$

$$2^{\log_2 n} = n^{\log_2 2} = n^1 = n$$

$$\Theta(n\log_2 n)$$

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

$$T\left(\frac{n}{2}\right) = 2\left[2T\left(\frac{n}{2^2}\right) + \left(\frac{n}{2}\right)^3\right] + n^3$$

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

$$T\left(\frac{n}{2^2}\right) = 2^2\left[2T\left(\frac{n}{2^3}\right) + \left(\frac{n^3}{2^2}\right)^3 \cdot \frac{1}{4}\right] + \frac{n^3}{2^2} + n^3$$

$$T\left(\frac{n}{2^2}\right) = 2^3\left[2T\left(\frac{n}{2^3}\right) + \frac{n^3}{2^4} + \frac{n^3}{2^2} + n^3\right]$$

$$\vdots$$

k steps

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + n^3\left(\frac{1}{2^0} + \frac{1}{2^2} + \frac{1}{2^4} \cdots \frac{1}{2}\right)$$

$$\hookrightarrow n^3\left(\frac{1}{4^0} + \frac{1}{4^1} + \cdots + \frac{1}{4^{k-1}}\right)$$

$2^0, 2^2, 2^4$

$\downarrow \quad \downarrow$

$4^0, 4^1, 4^2 \cdots 4^{k-1}$

If no Log $P=0$

$$2^x = 8 \rightarrow 2^3 = 8$$

$t(n) = 8 + \left(\frac{n}{2}\right) + n^3$

Master theorem

$a = 8$

$b = 2$ $\quad \log_6 a = \log_2 8 = 3$

$k = 3$ $\quad \Theta(n^3 \log n)$

$P = 0$

$t(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n} \rightarrow n\log^{-1} n$

$a = 2$

$b = 2$ $\quad \log_6 a = \log_2 2 = 1 = k$

$K = 1$

$P = -1$ $\quad \Theta\left(n \log(\log n)\right)$

$t(n) = 8t\left(\frac{n}{4}\right) + n^2 + 1$

$4^2 = 16$

$a = 8$

$b = 4$ $\quad \log_4 8 < k$ $\quad 1 \leq \log_4 8 \leq 2$

$k = 2$ $\quad \Theta(n^2 \log^0 n)$

$P = 0$ $\quad \Theta(n^2)$

$t(n)$ $3T\left(\frac{n}{4}\right) + n$

$a = 3$ $\quad \log_4 3 < k$

$b = 4$ $\quad \Theta\left(n^k \log^P n\right)$

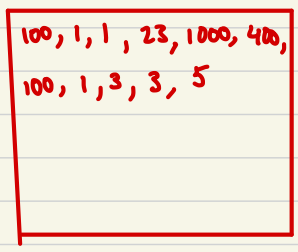
$k = 1$ $\quad \Theta(n)$

$P = 0$

```
a=[ ]
 for (i=1, i<10, i++)
       Print(....)

    if a[1] == 3
         Print(....)
```

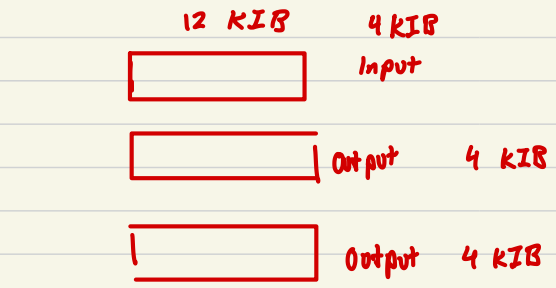
2^x KiB



KB = 1000 Bytes

KiB = 1024 Bytes

2^5 = 32 KIB



CSC510 Analysis of Algorithms
Divide and Conquer

Jose Ortiz
jortizco@sfsu.edu

Overview

- Approach
  - Divide and Conquer
- Divide and Conquer Algorithms
  - Binary Search
  - Merge Sort
  - Quick Sort
  - Matrix Multiplication
  - Closest Pair of Points

Divide

- Create sub problems of the main problem to focus in specific techniques that will help in solving the main problem

- Small problems are easier to handle, and solve than bigger ones

Conquer

- Recursively doing the same on each part until small (manageable and solvable) size is reached

- Combine the sub-solutions to form a solution to the original problem

Divide and Conquer Algorithms
Binary Search

Binary Search

- Pros:
  - Binary search algorithm uses the divide technique to reject parts/blocks of the data structure, that based on a specific condition, are not needed for the next iteration of the algorithm
  - Binary search algorithm follows a recursive pattern
  - Theta time complexity of binary search is $\Theta(\log n)$
  - Faster than many other recursive algorithms.
- Cons:
  - Data must be initially sorted before applying the algorithm

Binary Search Algorithm Example

Goal: does value 320 exist in the array? If so, return its index

Index	0	1	2	3	4	5	6	7	8	9
Values	24	32	88	99	100	101	200	300	320	350

middle = start + (end - start) // 2

0 + (6 - 0) // 2

0 + 6 // 2

0 + 3

Binary Search
Recursive Pseudocode and Complexity Analysis

```
function BS ( List , StartIndex , endIndex , target )
    Assert : ( StartIndex and endIndex ) ≥ 0
    IF STARTINDEX > ENDINDEX THEN
        RETURN -1 Because target was not found
    ENDIF
    MIDPLEINDEX ← ( STARTINDEX + ( endIndex - startIndex ) // 2 )
    TMPELEMENT ← the element found at middle INDEX
    IF TMPELEMENT = target then
        return MiddleIndex
    ENDIF
    IF TMPELEMENT < target then
        RETURN BS ( List , middleIndex + 1 , endIndex , target )
    ENDIF
    RETURN BS ( List , StartIndex , middleIndex - 1 , Target )
```

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

$a = 1 \quad k = 0$

$b = 2 \quad p = 0$

$$\log_b a = \log_2 1 = 0 = k$$

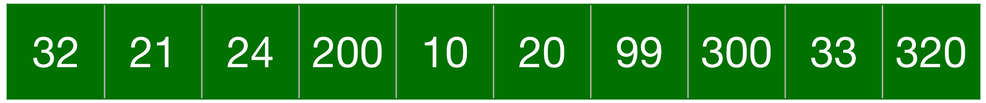
$$\Theta\left(n^k \log^{p+1} n\right)$$

Divide and Conquer Algorithms
Merge Sort

Merge Sort

- Pros:
  - Implements divide and conquer techniques
  - Like Binary Search algorithm, by nature, Merge Sort also follows a recursive pattern.
  - Time complexity of merge sort (worst, average and best cases) is $O(nlogn)$
  - Finds the middle, and divides the array in half. Then call itself until it merges the values
- Cons:
  - Typical implementations have Space Complexity $O(n)$. However, non-typical implementations (also stable) can be done with constant space complexity (in place)

Merge Sort Example



Merge Sort
Recursive Pseudocode and Complexity Analysis

function mergeSort (List)

If the List is empty or it has only one element then

RETURN : List

MiddleIndex ← (Length List) // 2)

LeftList ← Mergesort (a list with all the elements from Idex 0 to the middle Index) $T(\frac{n}{2})$

Rightlist ← Mergesort (a list with all the elements from middle Index +1 to the Last Index) $T(\frac{n}{2})$

Return Merge (Left list, Right List) ← $\Theta(n)$ $n^1 \text{Log}^0 n = n$

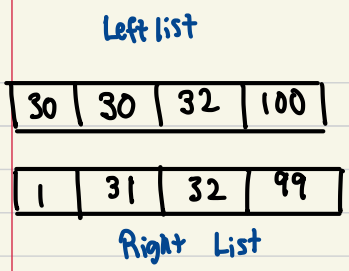
$T(n) = 2T(\frac{n}{2}) + n$

$a = 2 \quad k = 1$

$b = 2 \quad p = 0$

$\text{Log}_b a = \text{Log}_2 2 = 1 = k$

$2.1 \rightarrow \Theta(n^k \text{Log}^{p+1} n) \rightarrow \Theta(n^1 \text{Log}^1 n) \rightarrow \Theta(n \text{Log} n)$



Function Merge (LeftList, RightList)

n Initialize : newList ← of size Leftlist + Rightlist

2 Initialize : left list pointer ← 0 , Right list Pointer ← 0

2 Initialize : Length of Left list ← Len(Left list), Length of Right list ← Len(Right List)

n While (Left List pointer < Length of Left List and Right List pointer < Length of Right List)

θ(n)

IF (element at left list pointer ≤ element at Right List Pointer)

Newlist ← Element at left list pointer

Increment : Left list pointer + 1

Else :

Newlist ← element at Right pointer

Increment : Right List Pointer + 1

End While

n While (left list pointer < Length of Left List)

n While (Right list pointer < Length of Right List)

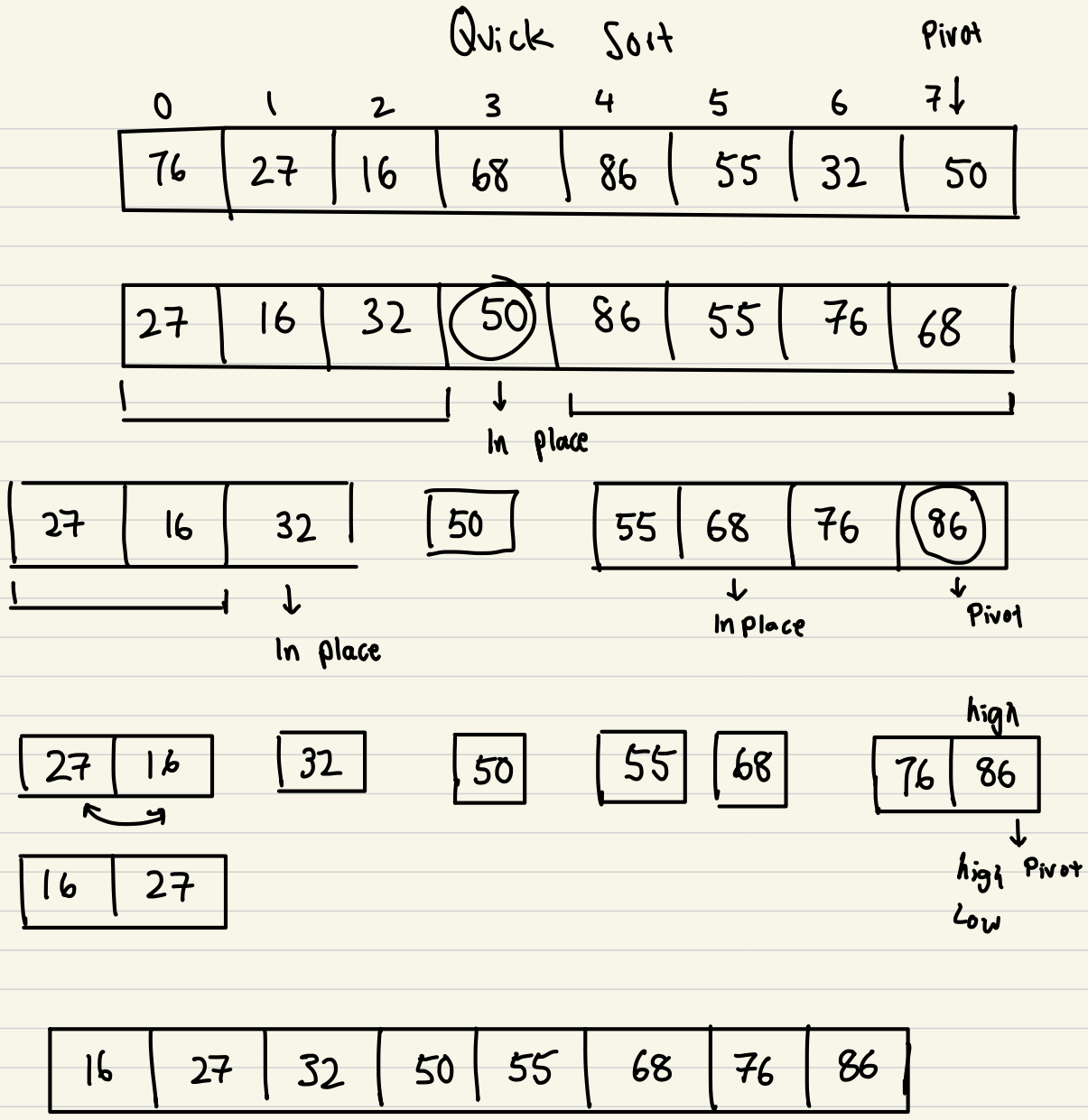
32 KIB
100, 3, 3, 5, 5, 5,
5, 5, 5, 6, 1000
4 KIB
Input
4 KIB
Input
4 KIB
Output
TempFile 1
4KIB
8 KIB
TempFile 2

Divide and Conquer Algorithms
Quick Sort

Quick Sort

- Pros:
  - Implements divide and conquer techniques (like merge sort). Is the most used algorithm to sort data
  - Like Merge Sort, quick sort also follows a recursive pattern.
  - Time complexity of merge sort in best and average cases is $O(nlogn)$ and the worst case is $O(n^2)$. Note that the worst case rarely happens
  - Works around a chosen pivot, and recursively, creates partitions in place
  - Depending on the task, sometimes, better than Merge Sort since Quick Sort space complexity is constant $O(1)$ (algorithm is implemented in place)
- Cons:
  - Not stable
  - Many different versions where pivot is initially selected from different indexes. (not really a con, but…)

Quick Sort



Merge sort

If high is Lower than 50 then

Low +1 Index and high swaps

16	27	32	50	55	68	76	86

Quick Sort Example

```
Function Partition ( List, Start , end )
Initialize : Low ← (Start -1)
Initialize : Pivot ← (List element in List)
Initialize : high ← Start
For each index high to end -1 :
  If (ElementAt high < pivot)
     Increment : Low + 1
     Swap ( ElementAtLow , ElementAt high )
Increment : Low + 1
Swap( pivot , ElementAt Low )
  Return Low
```

O(n)

```
Function Quick sort ( List , Start , end )
   If Start ≤ end :
   IndexOld Pivot : Partition (List, Start, end)
     Quick sort (List , Start, IndexOldPivot -1 ) ⟶ t(n/2)
     Quick sort ( List, IndexOldPivot +1 , end) ⟶ t(n/2)
```

$t(n) = 2t(\frac{n}{2}) + n$

$a = 2$

$b = 2$

$k = 1$

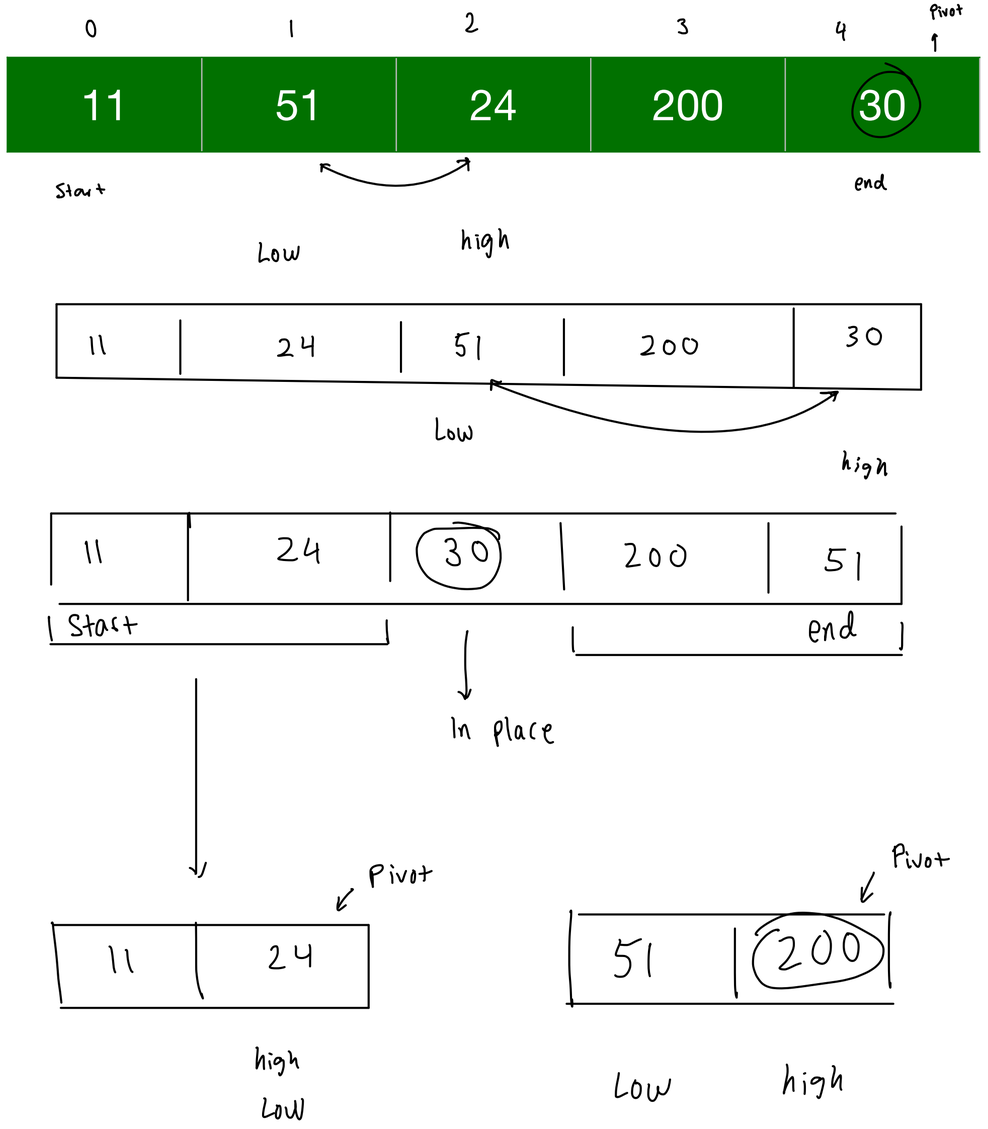
$P = 0$

$Log_b a = Log_2 2 = 1 = k$

Case 2.1

Ⓗ $(n^k Log^{P+1} n)$

Ⓗ $(n Log n)$



Sony WH-1000XM4

Quick Sort
Recursive Pseudocode and Complexity Analysis

```
quick_sort(array, start, end) {




}
```

```
partition(array, start, end) {




}
```

Divide and Conquer Algorithms
Closest Pair of Points

Quick Sort Worst, Best, Average Cases

- Worst Case: when pivot is the smaller or larger element. Which rarely happens, or only happens when the array is already sorted

$$T(n) = T(n-1) + n - 1;\ \ T(0) = 0;\ \ for\ n > 0$$

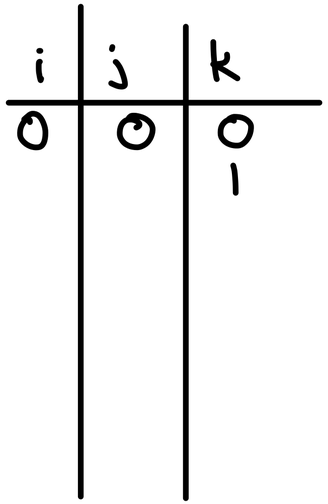
$$T(n) = \frac{n(n-1)}{2} = O(n^2)$$

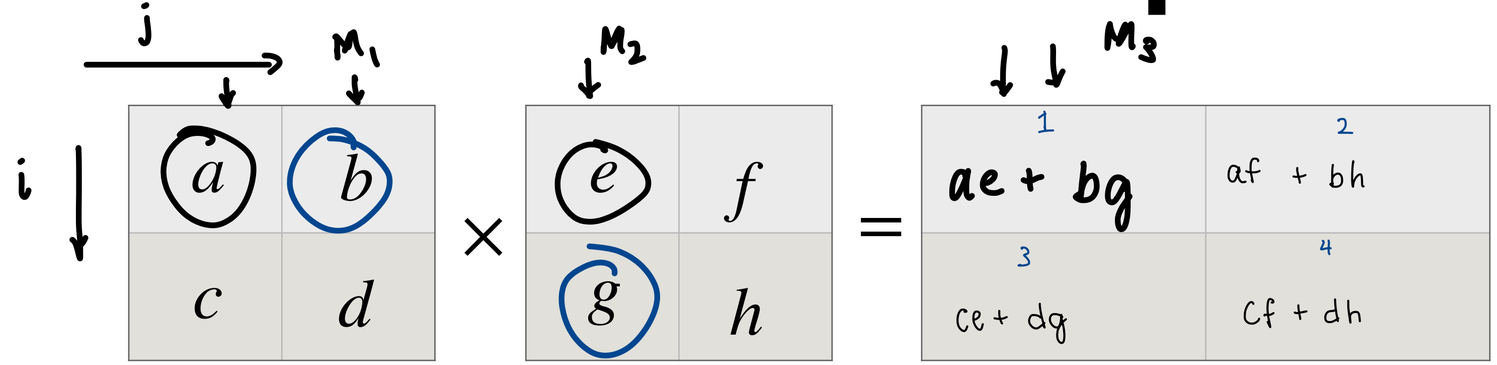
- Best and Average cases time complexities are $O(nlogn)$ like merge sort

Strassen's Algorithm

- In this topic, we'll cover the following approaches
  - Naive Matrix Multiplication Method
  - Divide and Conquer Matrix Multiplication Approach
  - Strassen's Matrix Multiplication Method.
  - Then, we'll analyze the time complexity and space complexity of such algorithms.

Naive Matrix Multiplication

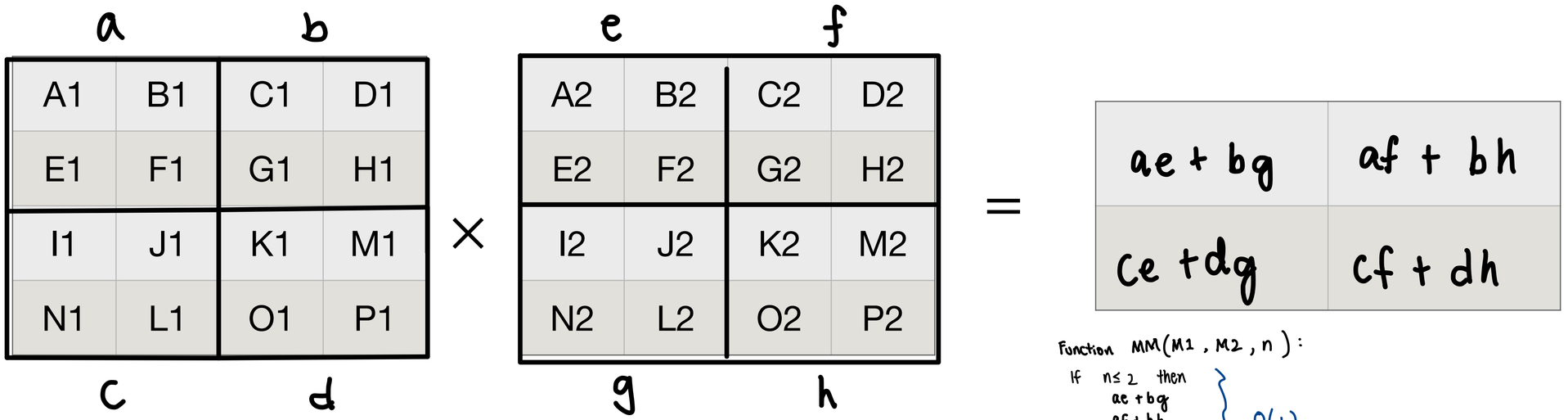





```
for(i=0, i<n, i++)
   for(i=0, j<n, j++)
      for(k=0, k<n; k++)
         M3[i][j] += M1[i][k] x M2[k][j]
                     M1[0][1] x M2[1][0]
                     b x g
```

$O(n^3)$

Divide and Conquer Matrix Multiplication



$T(n) = 8 + (\frac{n}{2}) + 4(n^2)$

$a = 8 \quad k = 2$

$b = 2 \quad P = 0$ Case 1

$\log_b a = \log_2 8 = 3 > K$

$O(n^{\log_b a}) = O(n^3)$

```
Function MM(M1, M2, n):
  If n ≤ 2 then                    } O(1)
      ae + bg
      af + bh
      ce + dg
      cf + dh
  n = n/2
√ MM( a,e ) + √MM( b,g )
√ MM( a,f ) + √MM( b,h )
√ MM( c,e ) + √MM( d,g )
√ MM( c,f ) + √MM( d,h )
```

Strassen's Method

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \times \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \overbrace{\begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 + p7 \end{bmatrix}}^{\begin{matrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{matrix}}$$

$$p1 = A(F - H) \quad 1$$

$$p2 = (A + B)H \quad 2$$

$$p3 = (C + D)E \quad 3$$

$$p4 = D(G - E) \quad 4$$

$$p5 = (A + D)(E + H) \quad 5$$

$$p6 = (B - D)(G + H) \quad 6$$

$$p7 = (A - C)(E + F) \quad 7$$

$T(n) = 7T(n/2) + n^2$

$a = 7 \qquad k = 2$

$b = 2 \qquad p = 0$

$\log_2 7 = 2^{x=3} = 7$

$x > 2$

$x < 3$

Closest Pair of Points

- Used to solve many computational geometric problems, as well as, many algorithms that require unsupervised data
- Time complexity using brute force is $O(n^2)$ which can be optimized to $O(nlog^2n)$, and then, $O(nlogn)$ using divide and conquer techniques.
- Widely used in natural language processes (search engines domain) and data science.

not going to be on final

Closest Pair of Points Example

1. Put all the points in an array sorted by x distance
2. Half the array, so each array (S_1, S_2) has size $l = n/2$
3. Find the closest distance between two points in each of the arrays: $d_1 \; and \; d_2$, recursively
4. Compute the minimum distance of $d_1 \; and \; d_2$ as $d = min(d_1, d_2)$. The distance between two points is:

$$||pq|| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

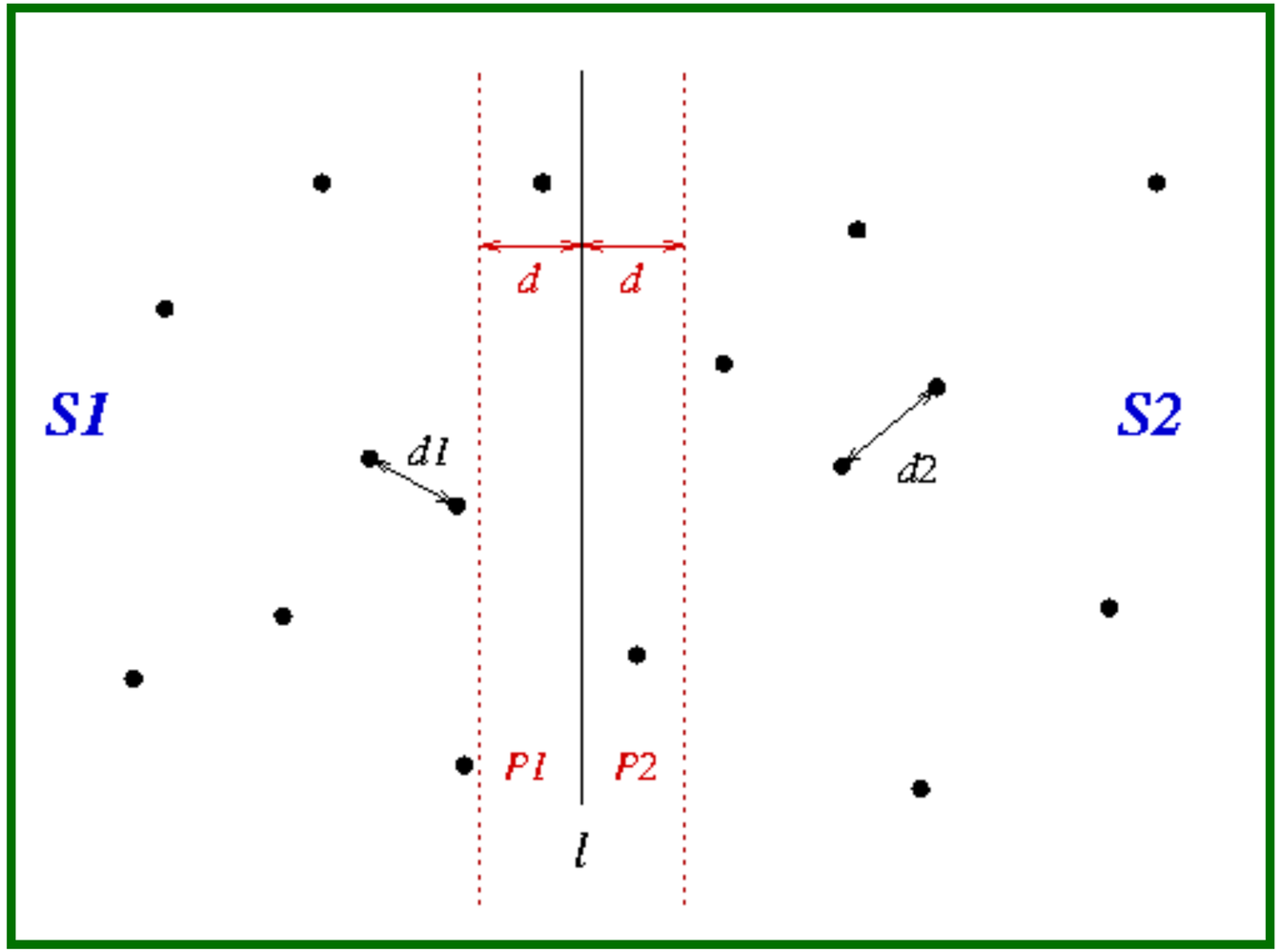


Image source: http://www.programming-algorithms.net/article/39650/Merge-sort

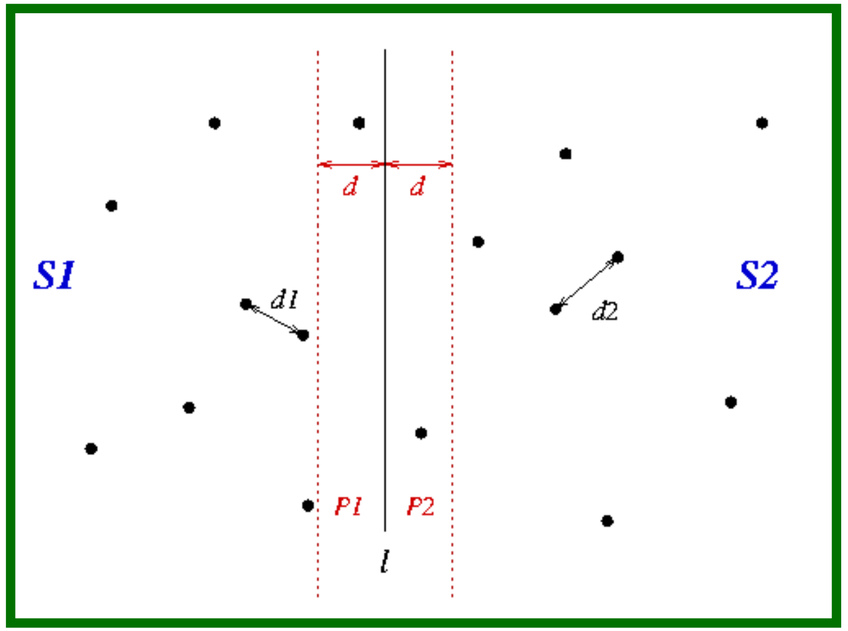
Closest Pair of Points Example

5. Eliminate points that lie farther than d apart from l (strip)

6. Sort the remaining points by their $y - coordinate$

7. Compute the distance of points from step 6 to its x closest neighbors.

8. If any of those distances is less than d, then update d



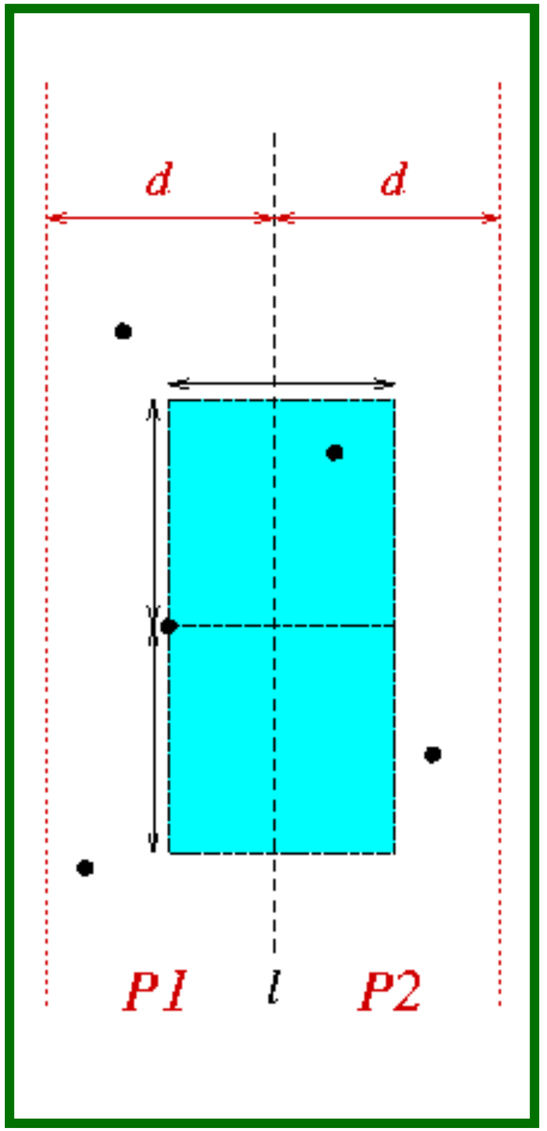
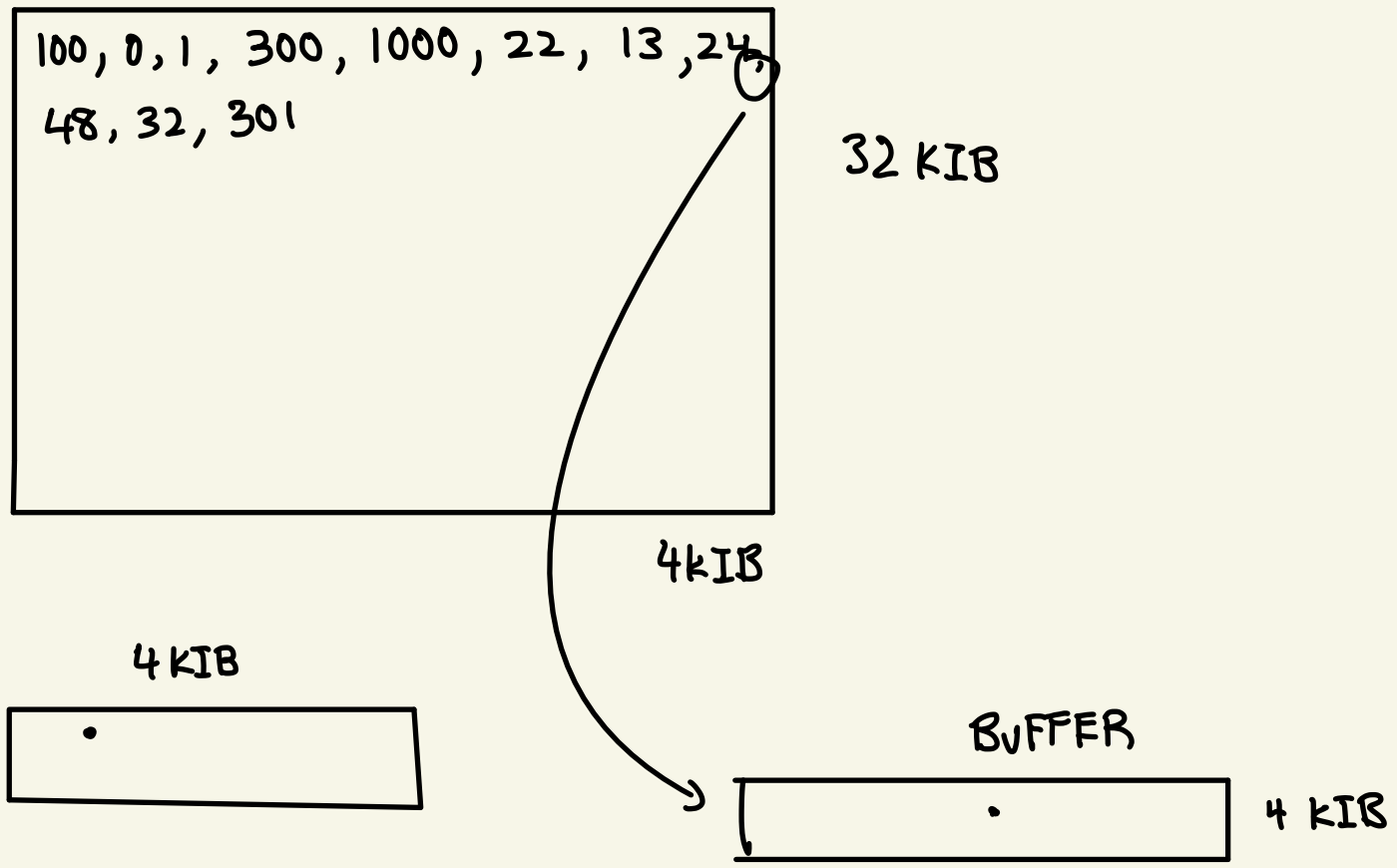


Image source: http://www.programming-algorithms.net/article/39650/Merge-sort

Closest Pair of Points Time Complexity

- Step 4 of this algorithm (sort) is $O(nlogn)$. However, for each recursive call, we need to half the array by $l = n/2$. This will increase the time complexity to $O(logn * nlogn) = O(nlog^2 n)$
- Can it be further optimized to $O(nlogn)$? **YES!**
  - Steps 3 and 4 are slightly modified to become one single step:
    - Recursively compute the distance in $(S_1 \ and \ S_2)$ returning the points in each set sorted in order by y-coordinates
  - In step 6, we only need to merge $(S_1 \ and \ S_2)$ in $O(n)$
  - This optimizations in the algorithm will improve its time complexity to $O(nlogn)$

HW #2 Problem 2



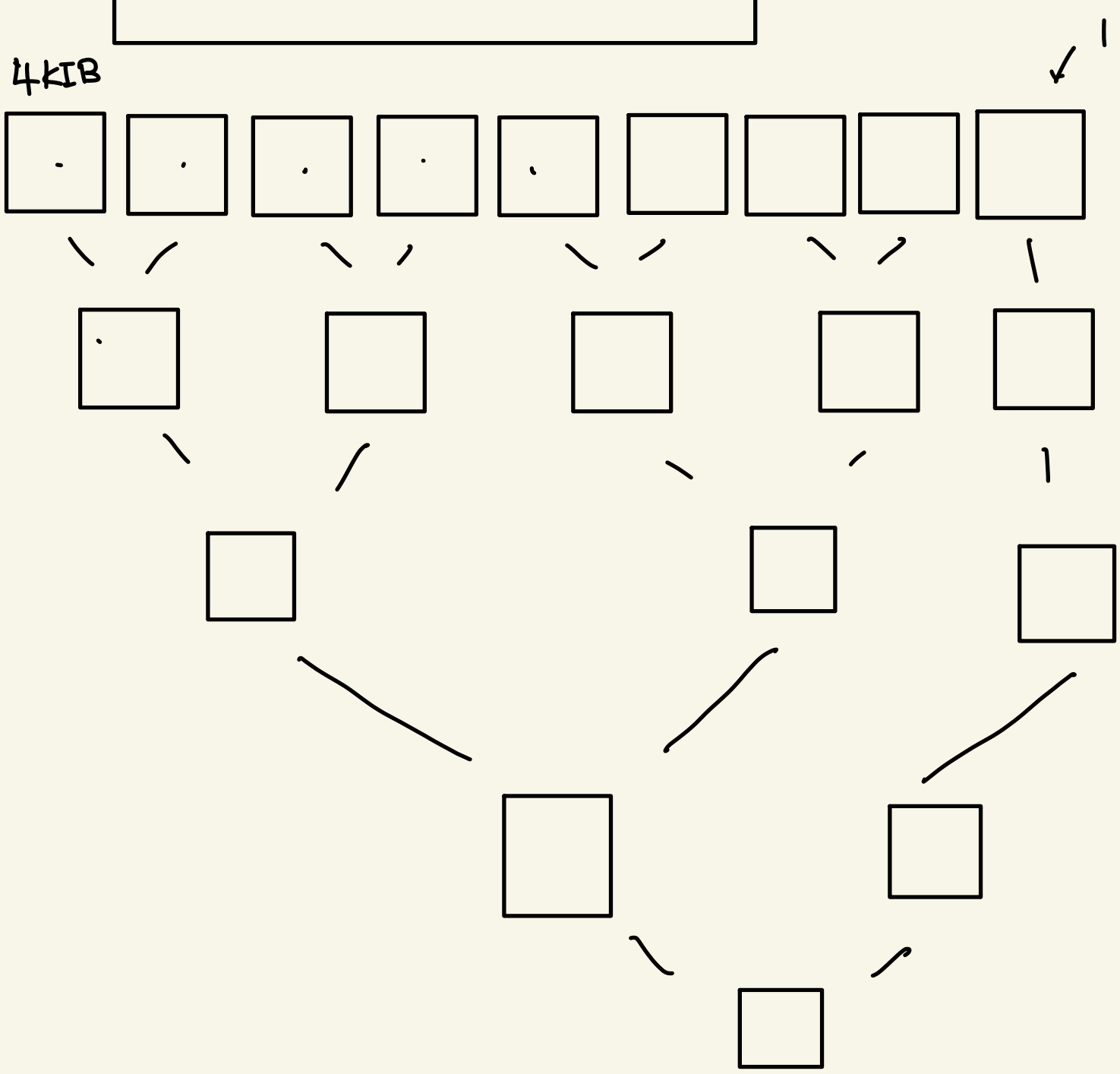
33 KIB

2^K

n → size

4KIB

1 KIB

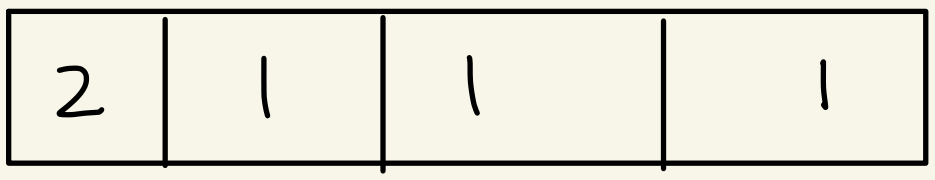


32 kIB

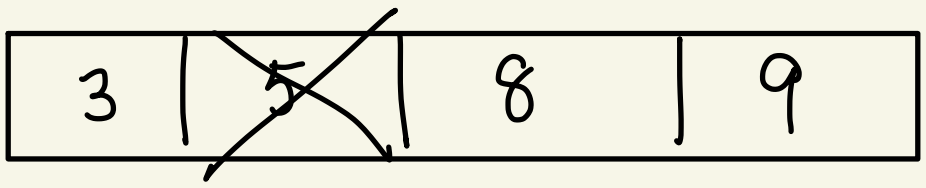
2, 1, 1, 1, 3, 5, 8, 9,
16, 15, 15, 100, 99, 30, 31,
29

Quick sort

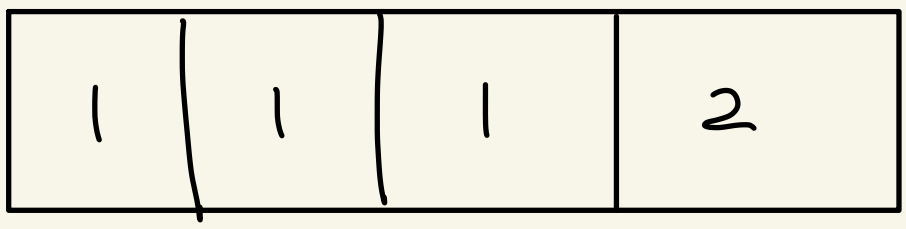
Input 1



Input 2



Output 1



F_1

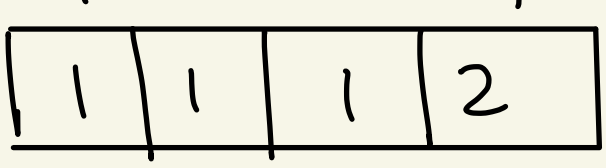
1, 1, 1, 2

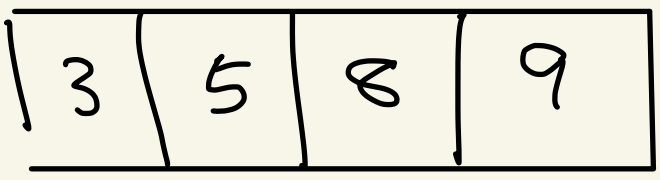
F_2

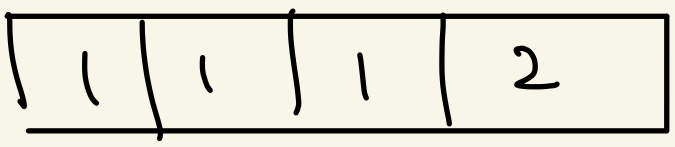
3, 5, 8, 9

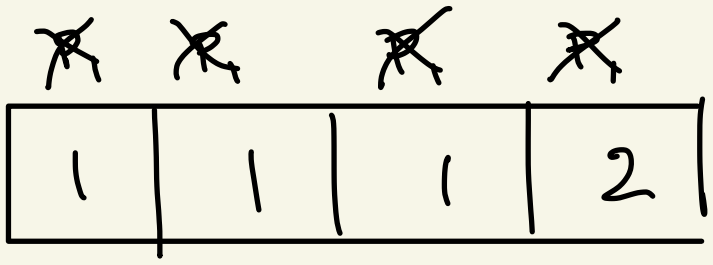
4 kIB

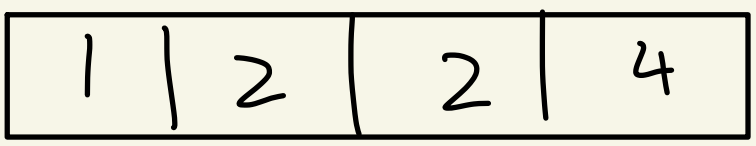
P_1 P_1 P_1 P_1

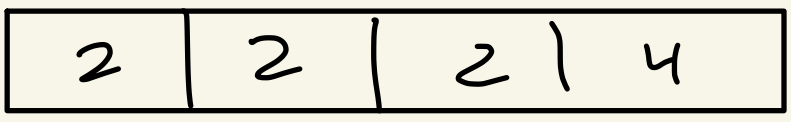












1, 1, 1, 2, 3, 5,
8, 9

8KIB

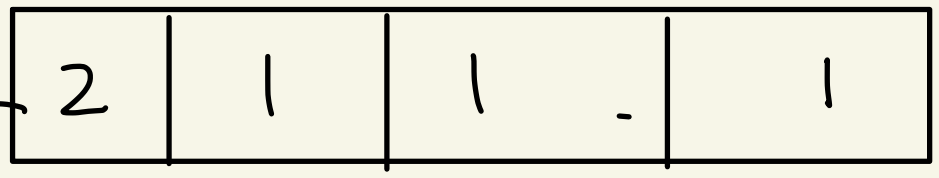
1, 2, 2, 4, 5,
5, 8, 8

32 kIB

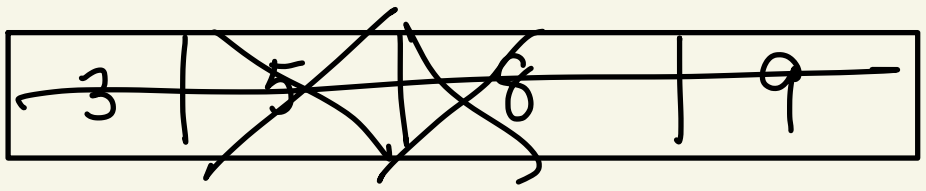
2, 1, 1, 1, 3, 5, 8, 9,
16, 15, 15, 100, 99, 30, 31,
29

Quick sort

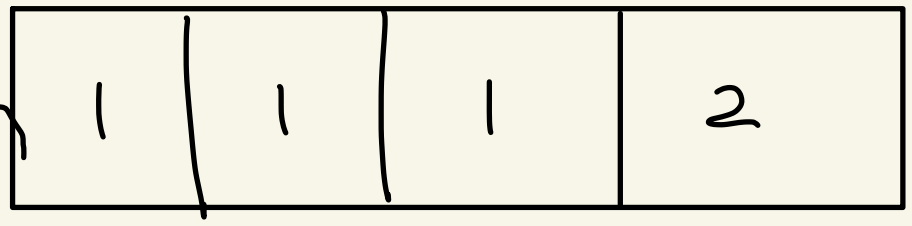
Input 1

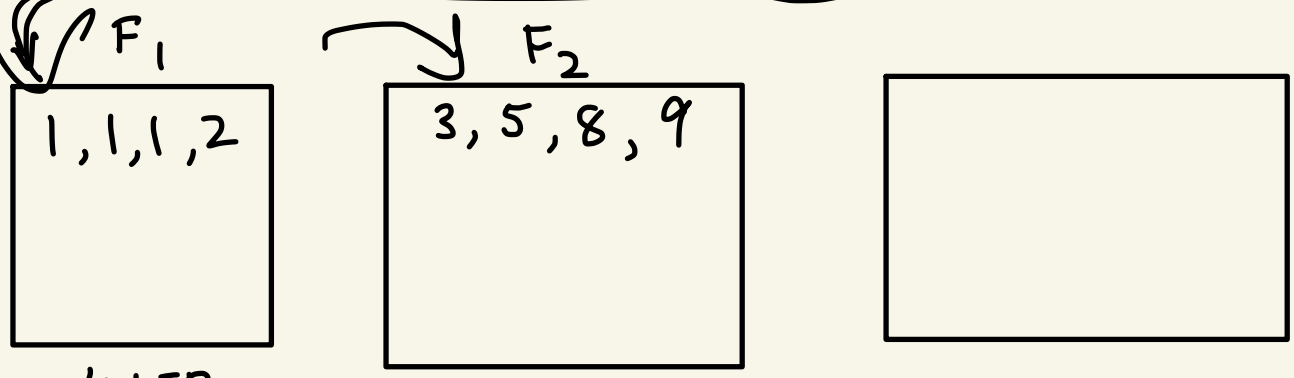


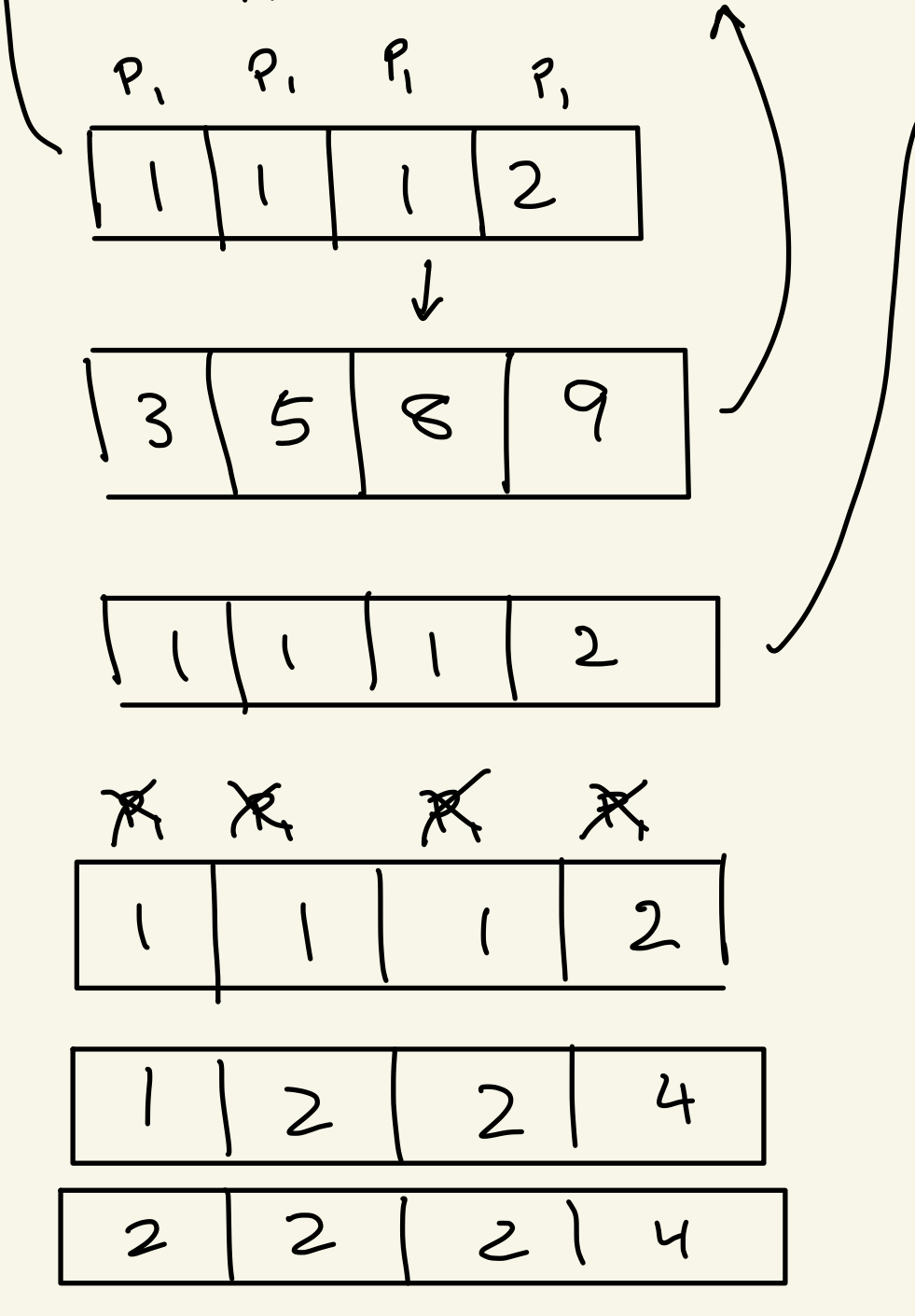
Input 2

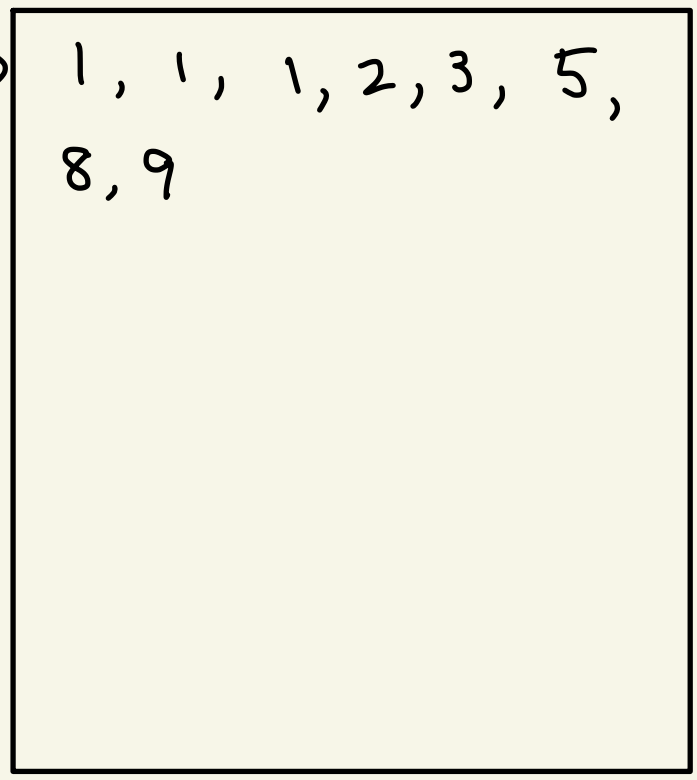


Output 1









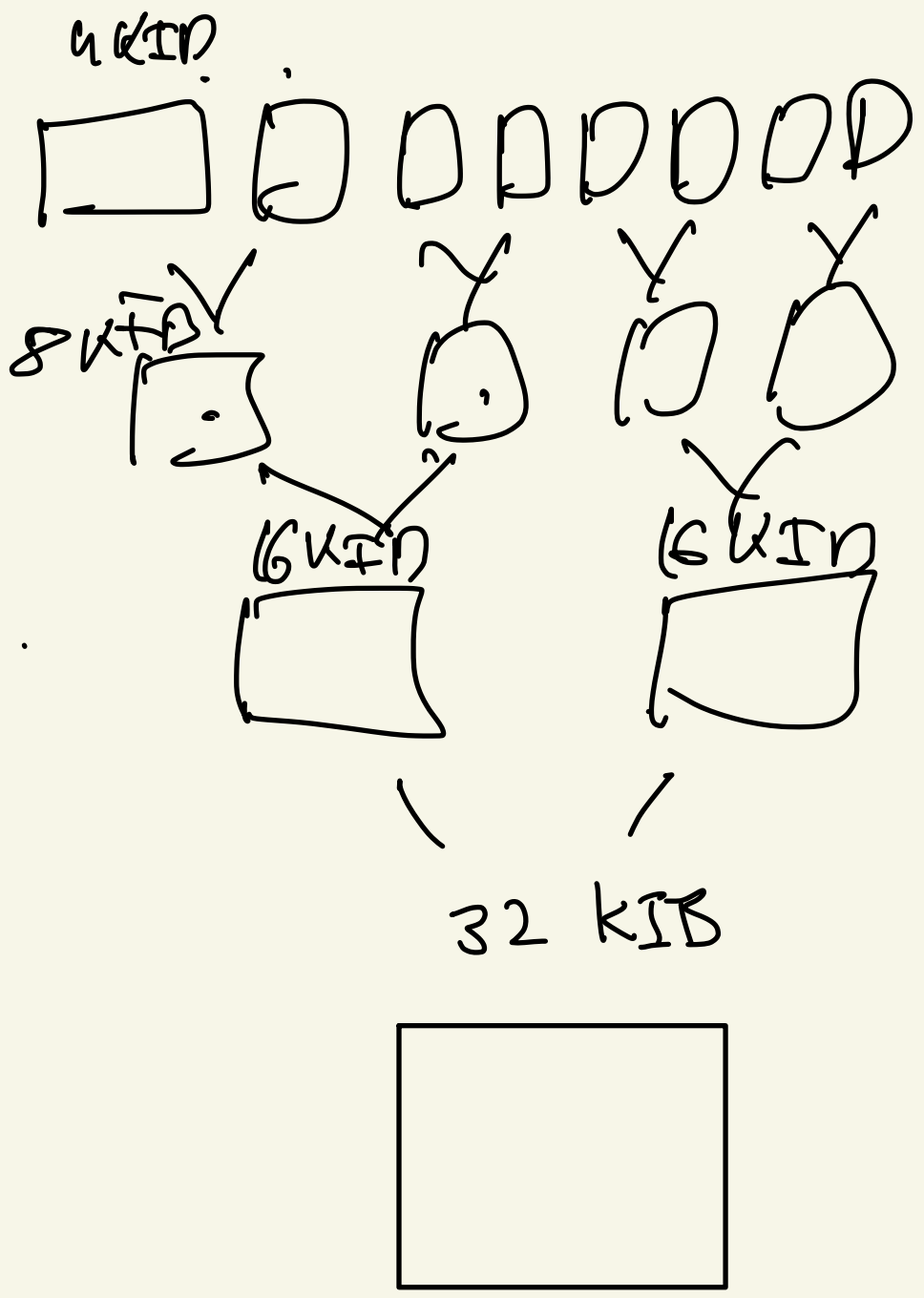
1, 2, 2, 4, 5,
5, 8, 8

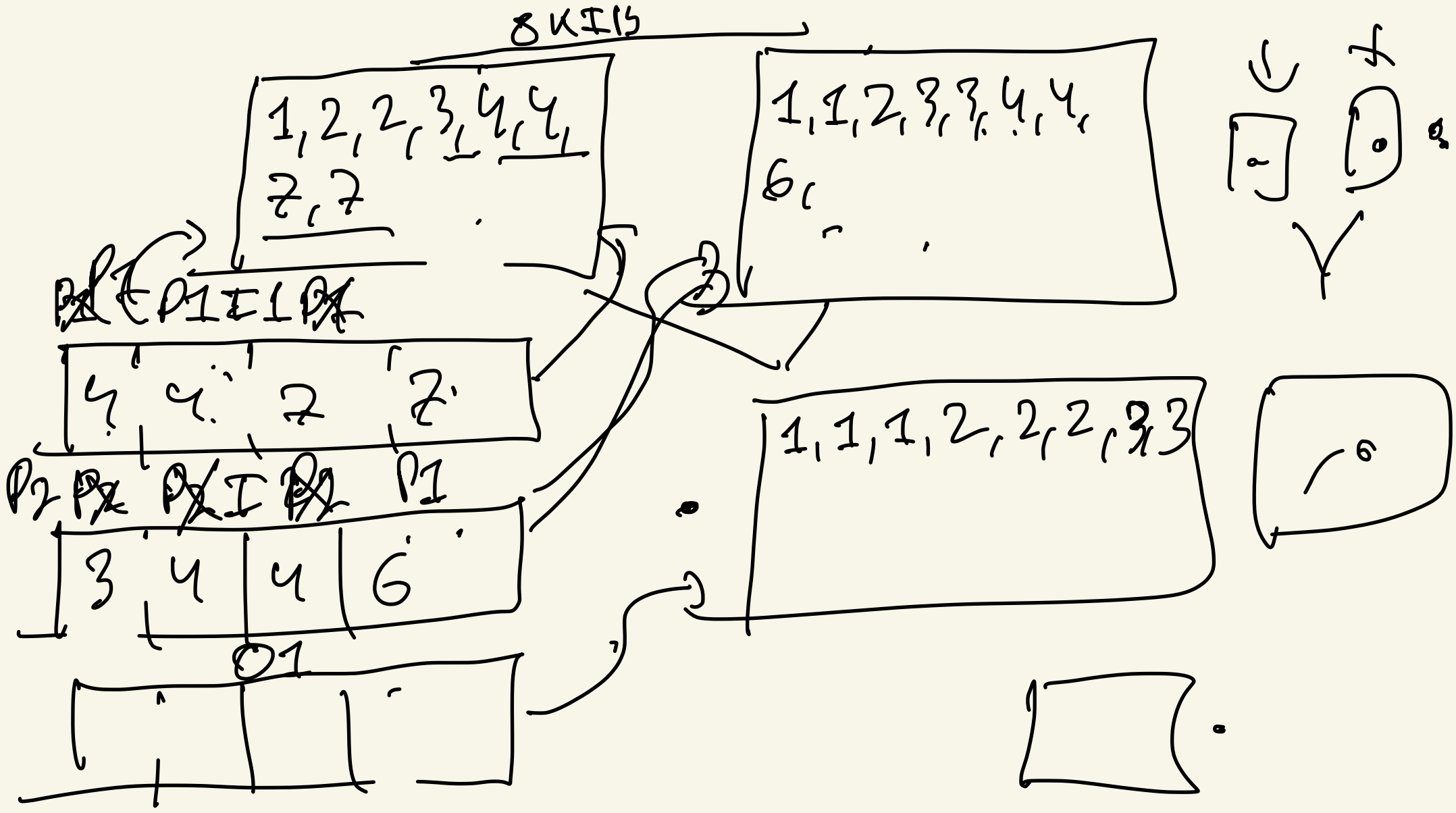
32 KiB

2, 1, 1, 1, 3, 5, 8, 9

16, 15, 15, 100, 99, 30, 31,

29





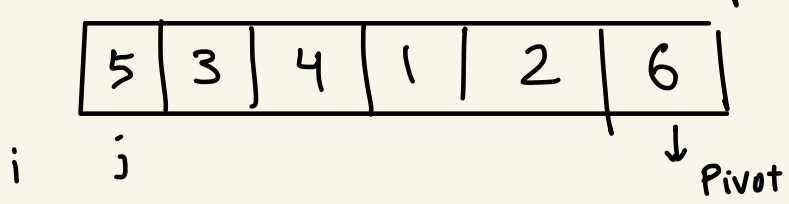
QuickSort $O(n^2)$

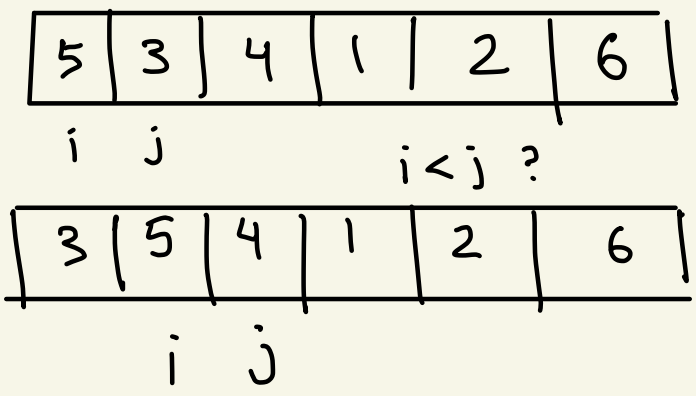
mergesort $O(n \log n)$

$T(n) = n\log n + n^2$

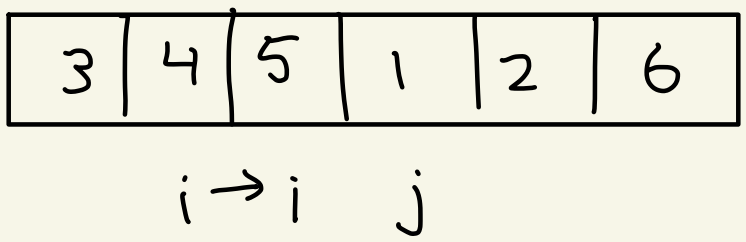
Quick sort example With 6 as highest

In place (highest element)

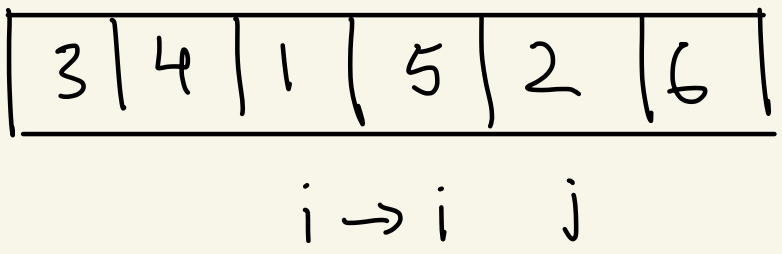




j < pivot ? ✓

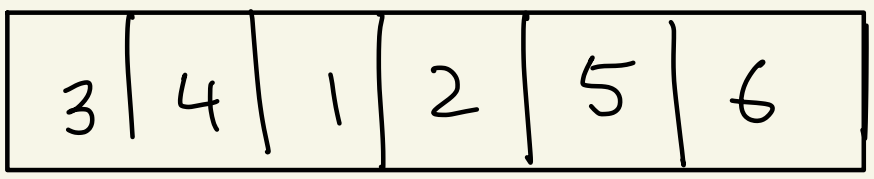


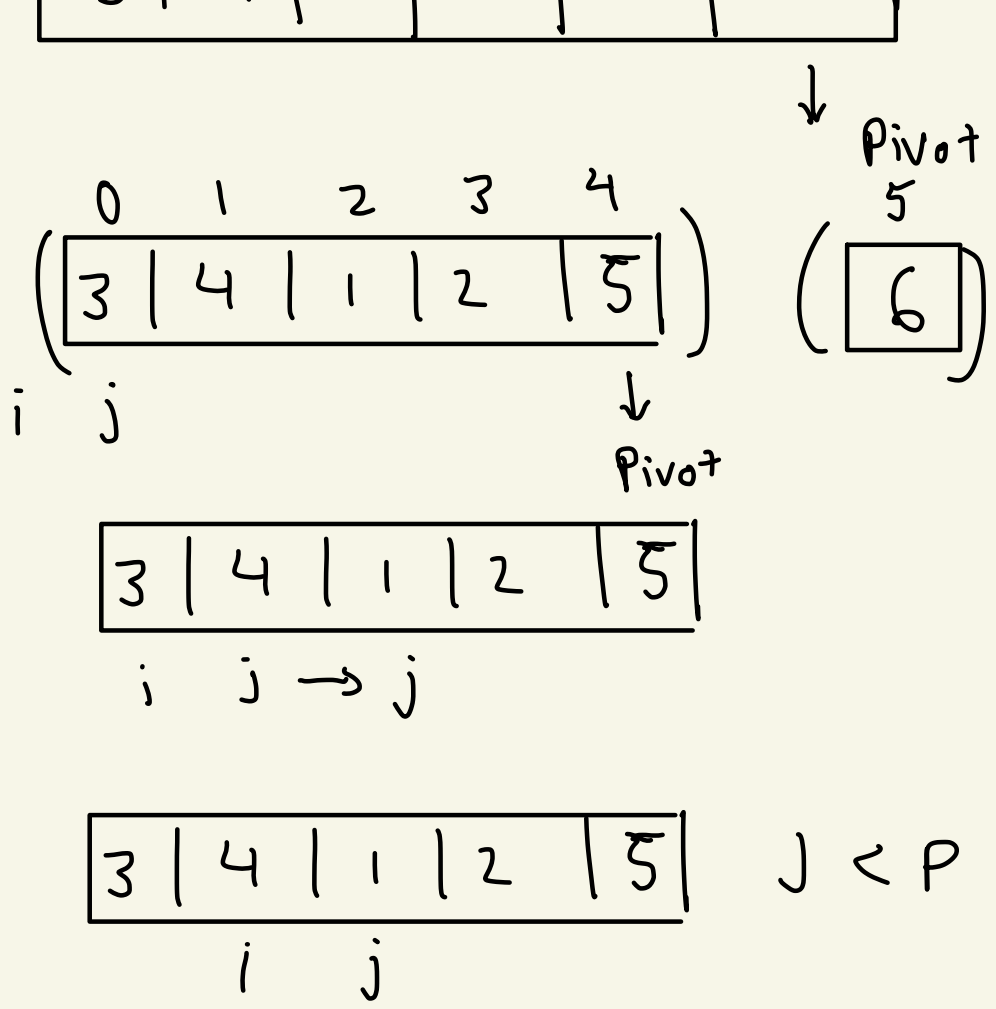
J < p ? ✓ Increment i +1



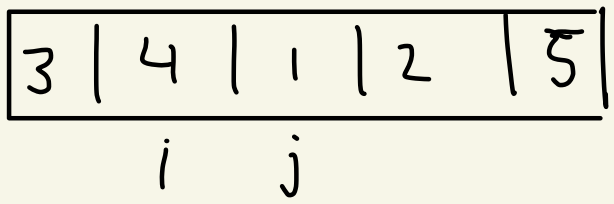
J < p ? ✓ Increment i +1

In Place

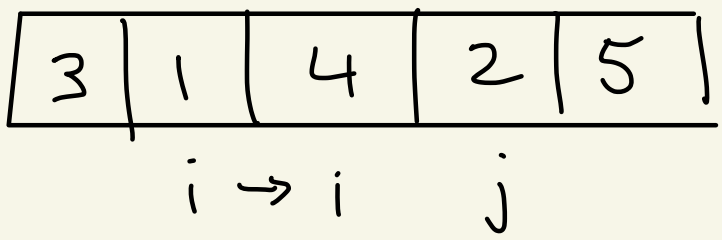




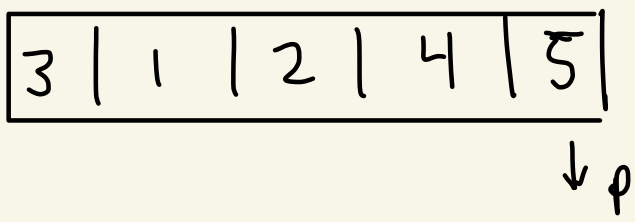
J < p ? ✓ Increment i +1



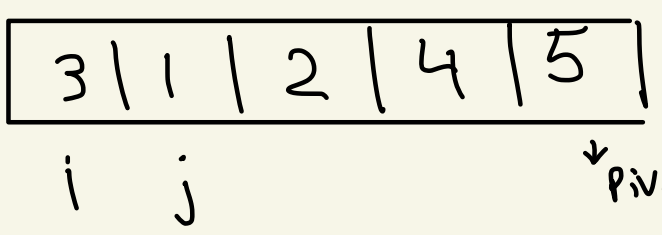
J < P ? ✓ Increment i + 1



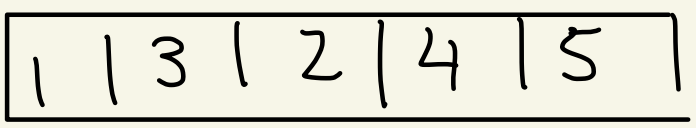
J < p ? ✓ Increment i + 1



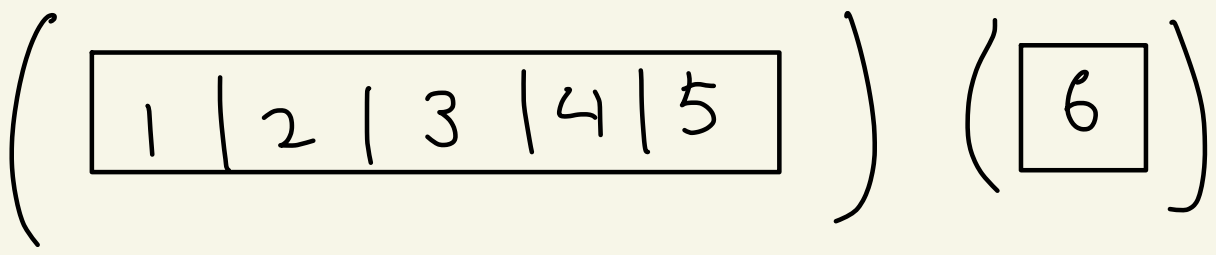
J < P ? ✓ Increment i + 1

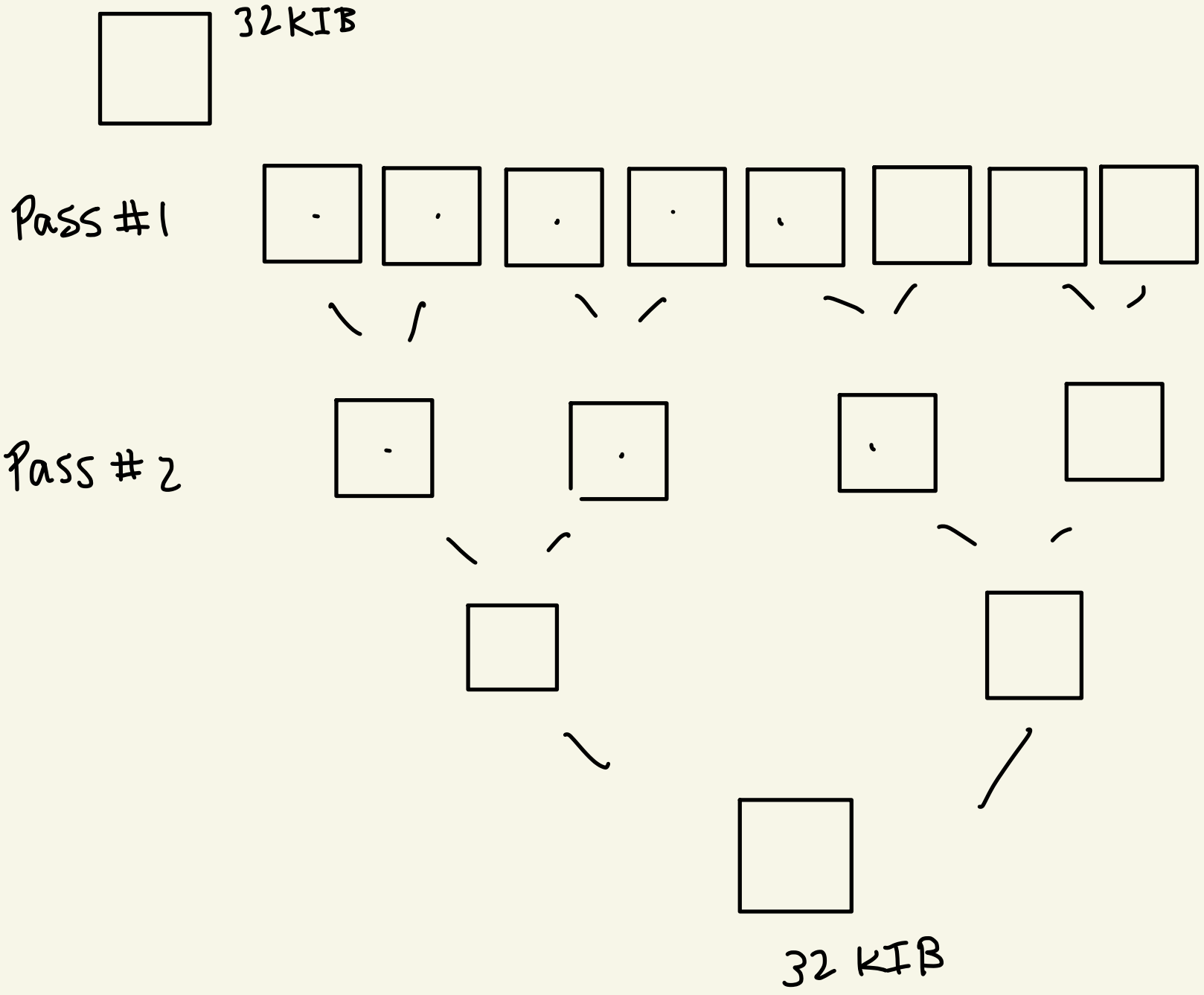


J < P ✓



J < P ✓




32KIB
Pass #1
Pass #2
32 KIB

CSC510 Analysis of Algorithms
Backtracking Algorithms

Jose Ortiz
jortizco@sfsu.edu

Overview

- Sum of Subsets
- N-Queens
- Graph Coloring

What are backtracking algorithms ?

- In a brute force approach, all the solutions (optimal and non-optimal) are computed and therefore, taken into consideration. On the other hand, in backtracking algorithms, all the solutions are computed, but abandoned if the solution cannot be completed.

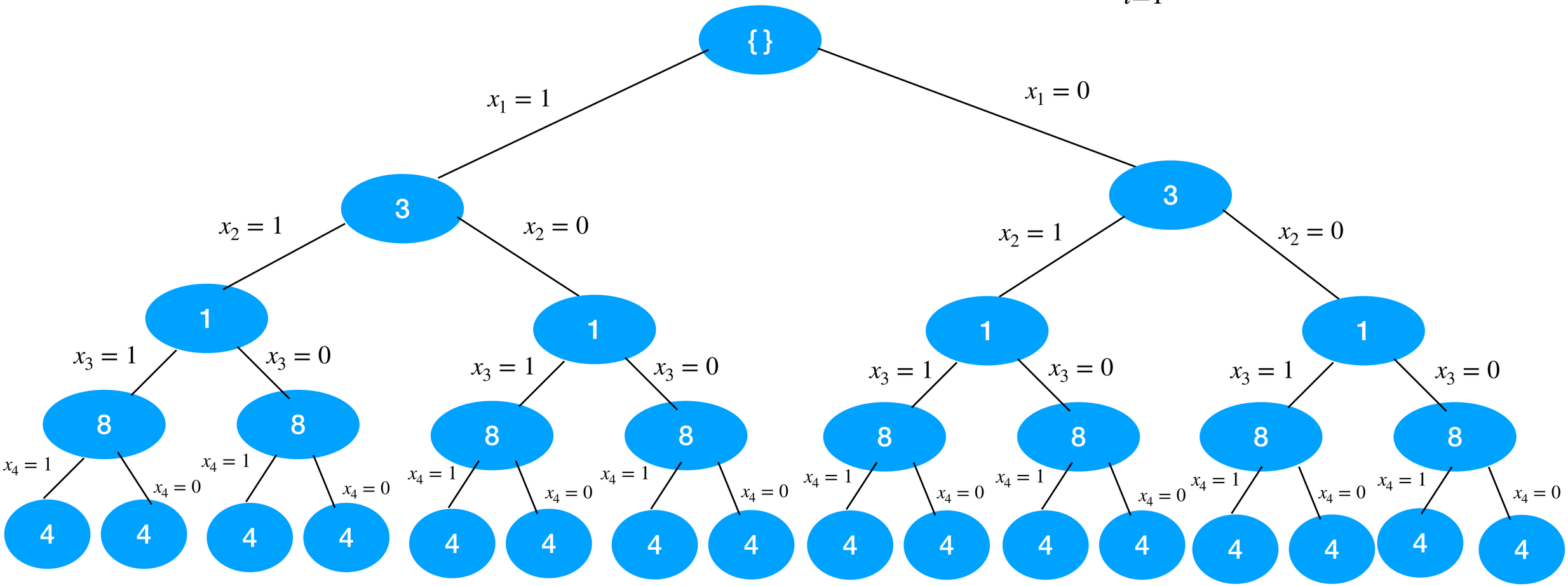
- Backtracking considers the concept of “partial candidate solution”. That’s it, it builds candidates to the solutions.

- In general, backtracking algorithms use the depth-first search method. However, more optimized backtracking approaches such as Branch and Bound use the best-case method.

Backtracking Sum of Subsets

Sum of Subsets Brute Force Approach

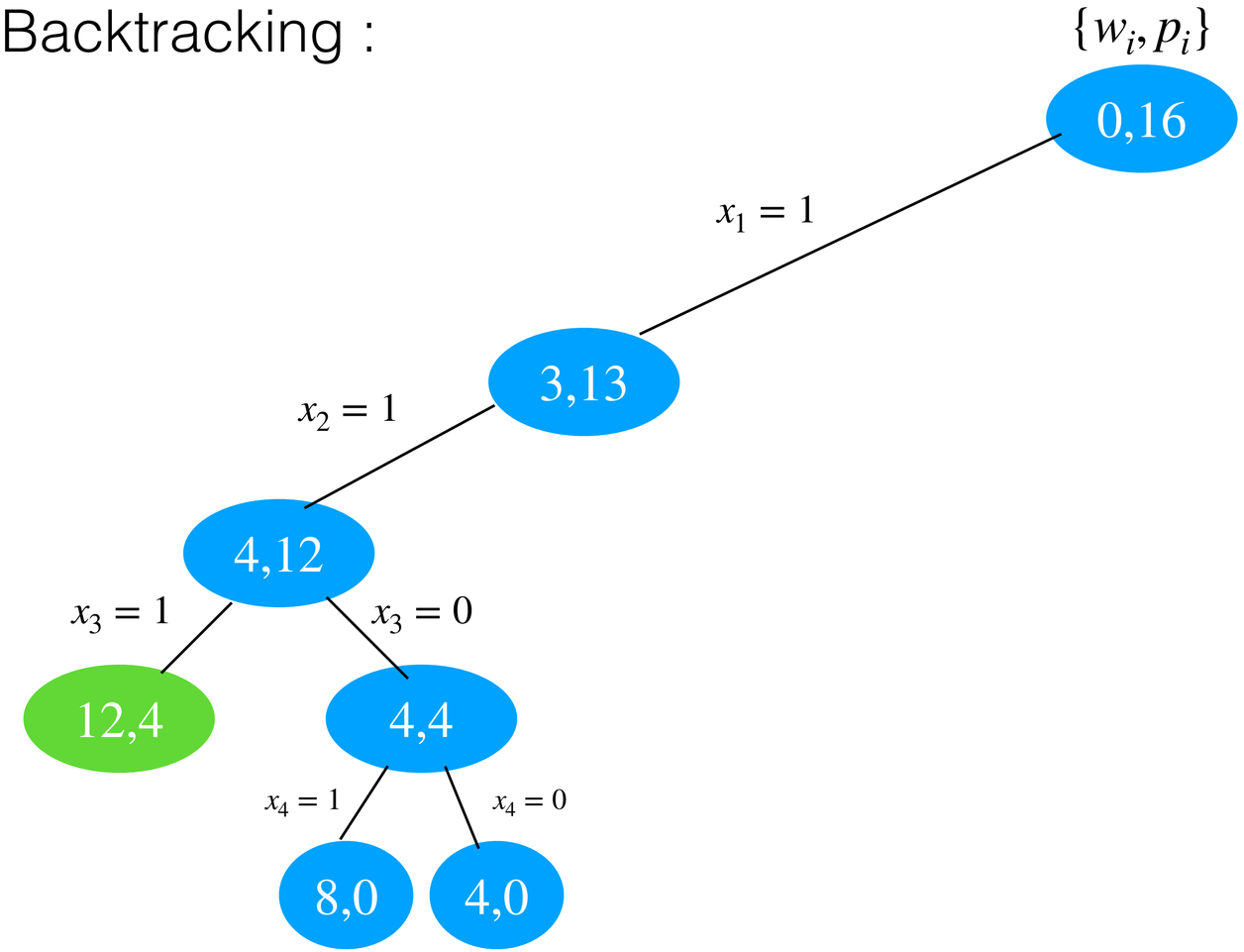
Given the following set of items. $S = \{3,1,8,4\}$ find all its subsets s where $\sum_{i=1}^{n} s_i = m;\ m = 12$



Sum of Subsets Backtracking

Given the following set of items. $S = \{3,1,8,4\}$ find all its subsets s where $\sum_{i=1}^{n} s_i = m;\ \ m = 12$

Backtracking :



Bounds

$$\sum_{i=1}^{n} w_i x_i \leq m \qquad \sum_{i=1}^{n} w_i p_i \geq 0$$

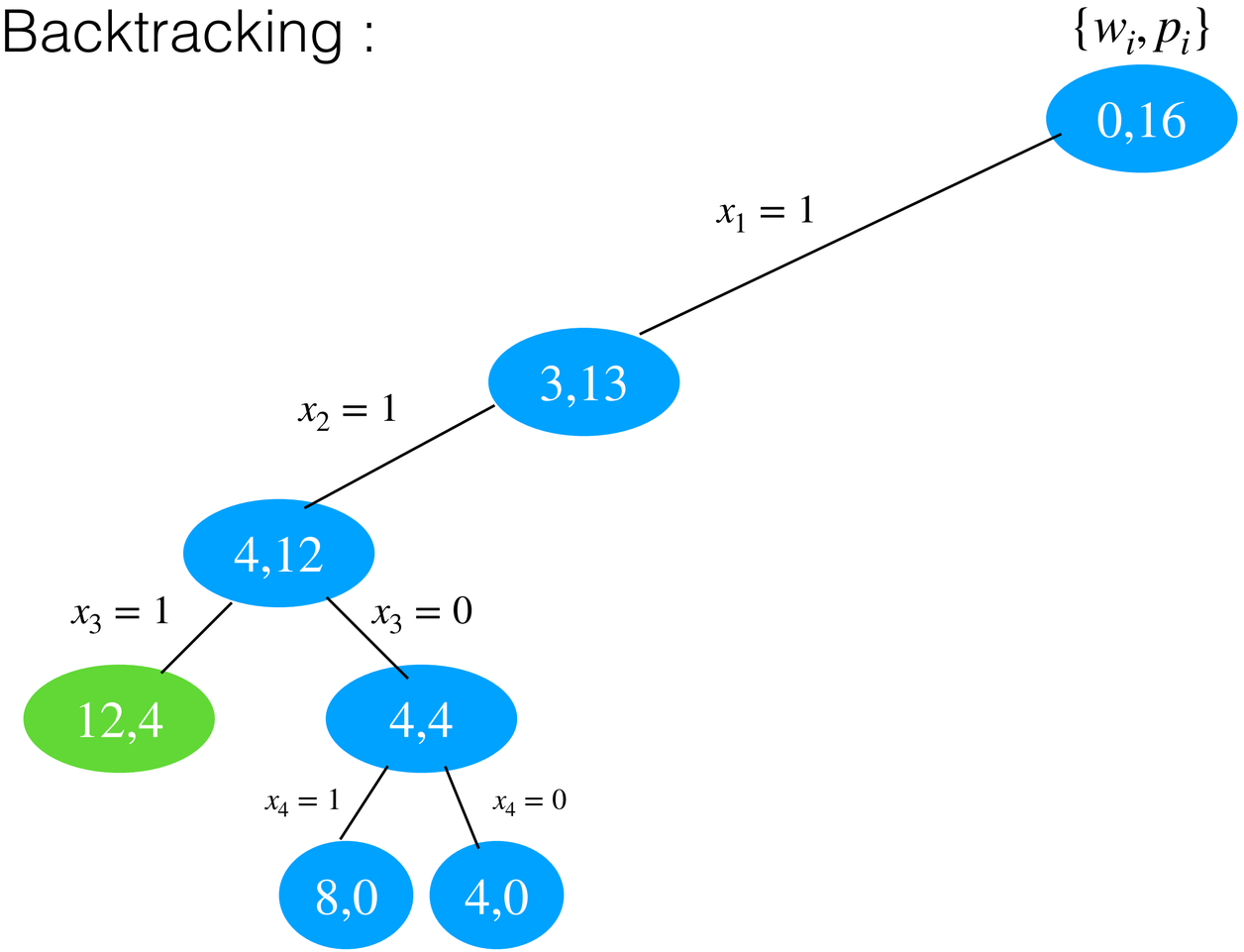
Operations

1. $x_i = 1$ *(included)*
   $x_i = 0$ *(not included)*
2. *Add* X_i *to* W_i
3. *Substract* X_i *from* P_i

Sum of Subsets Backtracking

Given the following set of items. $S = \{3,1,8,4\}$ find all its subsets s where $\sum_{i=1}^{n} s_i = m;\ \ m = 12$

Backtracking :



Bounds

$$\sum_{i=1}^{n} w_i x_i \leq m \qquad \sum_{i=1}^{n} w_i p_i \geq 0$$

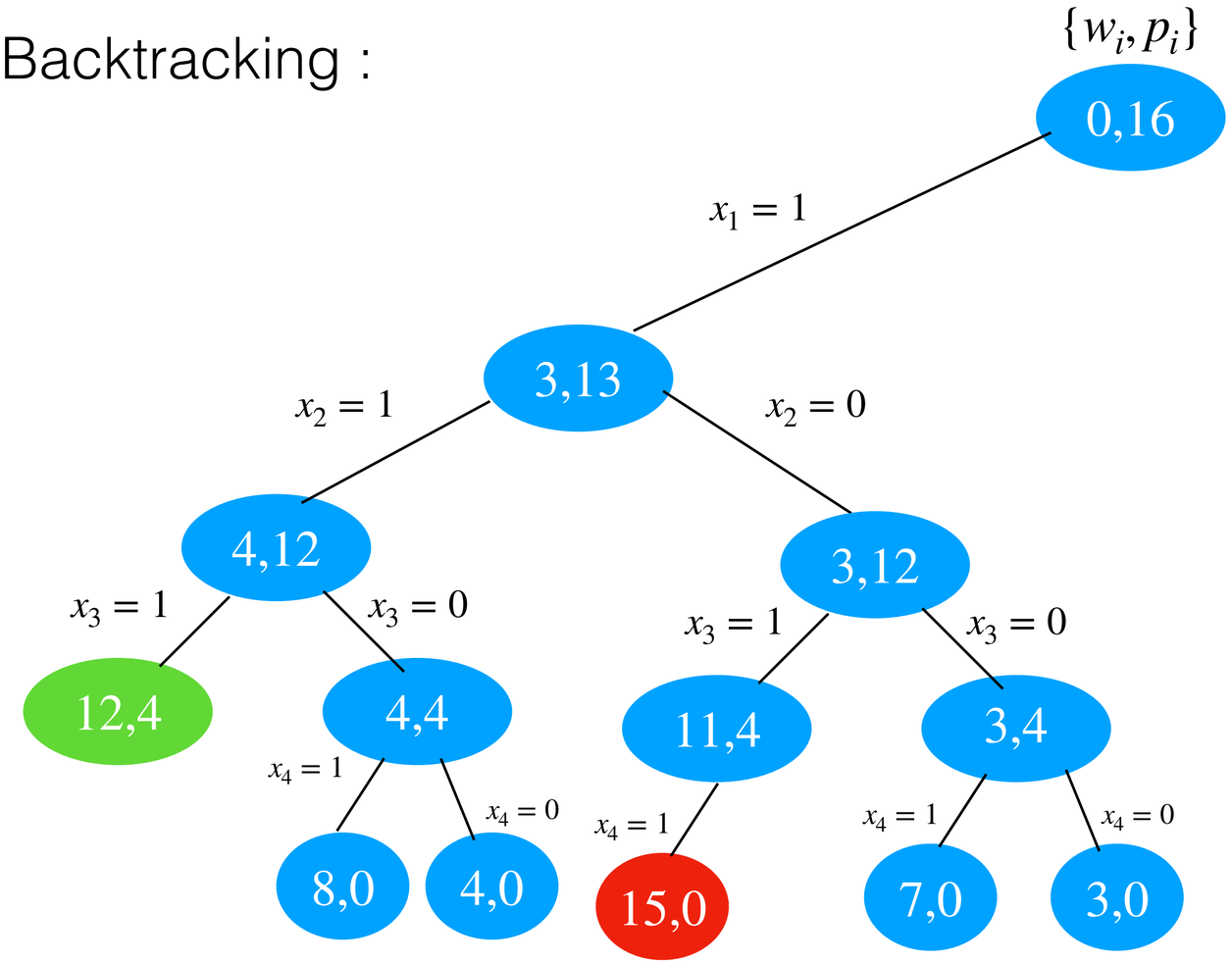
Operations

1. $x_i = 1$ *(included)*
   $x_i = 0$ *(not included)*
2. *Add* X_i *to* W_i
3. *Substract* X_i *from* P_i

Sum of Subsets Backtracking

Given the following set of items. $S = \{3,1,8,4\}$ find all its subsets s where $\sum_{i=1}^{n} s_i = m;\ \ m = 12$

Backtracking :



Bounds

$$\sum_{i=0}^{n} x_{i+1} \leq m \qquad \sum_{i=1}^{n} w_i p_i \geq 0$$

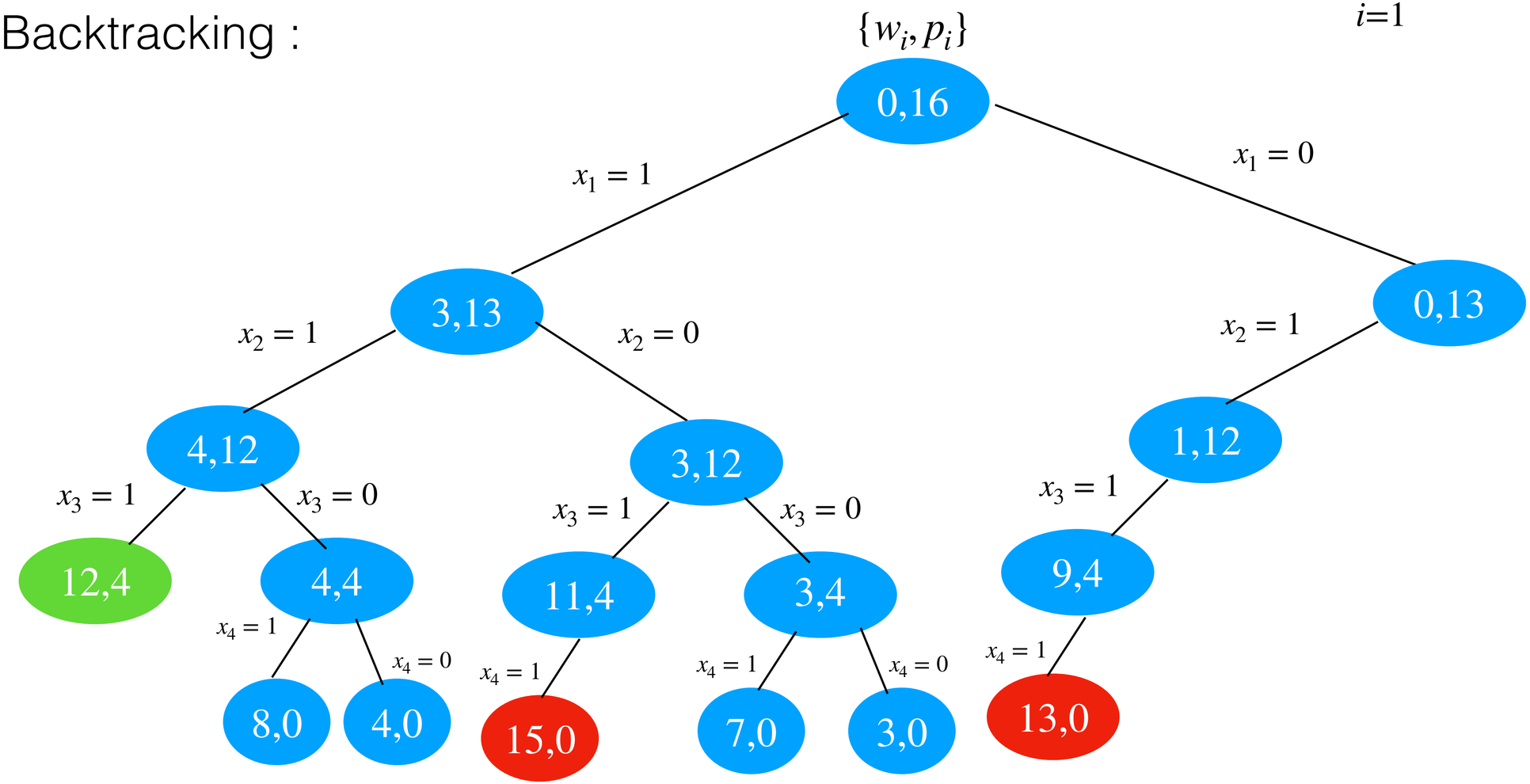
Operations

1. $x_i = 1$ *(included)*
   $x_i = 0$ *(not included)*
2. *Add* X_i *to* W_i
3. *Substract* X_i *from* P_i

Sum of Subsets Backtracking

Given the following set of items. $S = \{3,1,8,4\}$ find all its subsets s where $\sum_{i=1}^{n} s_i = m;\ \ m = 12$

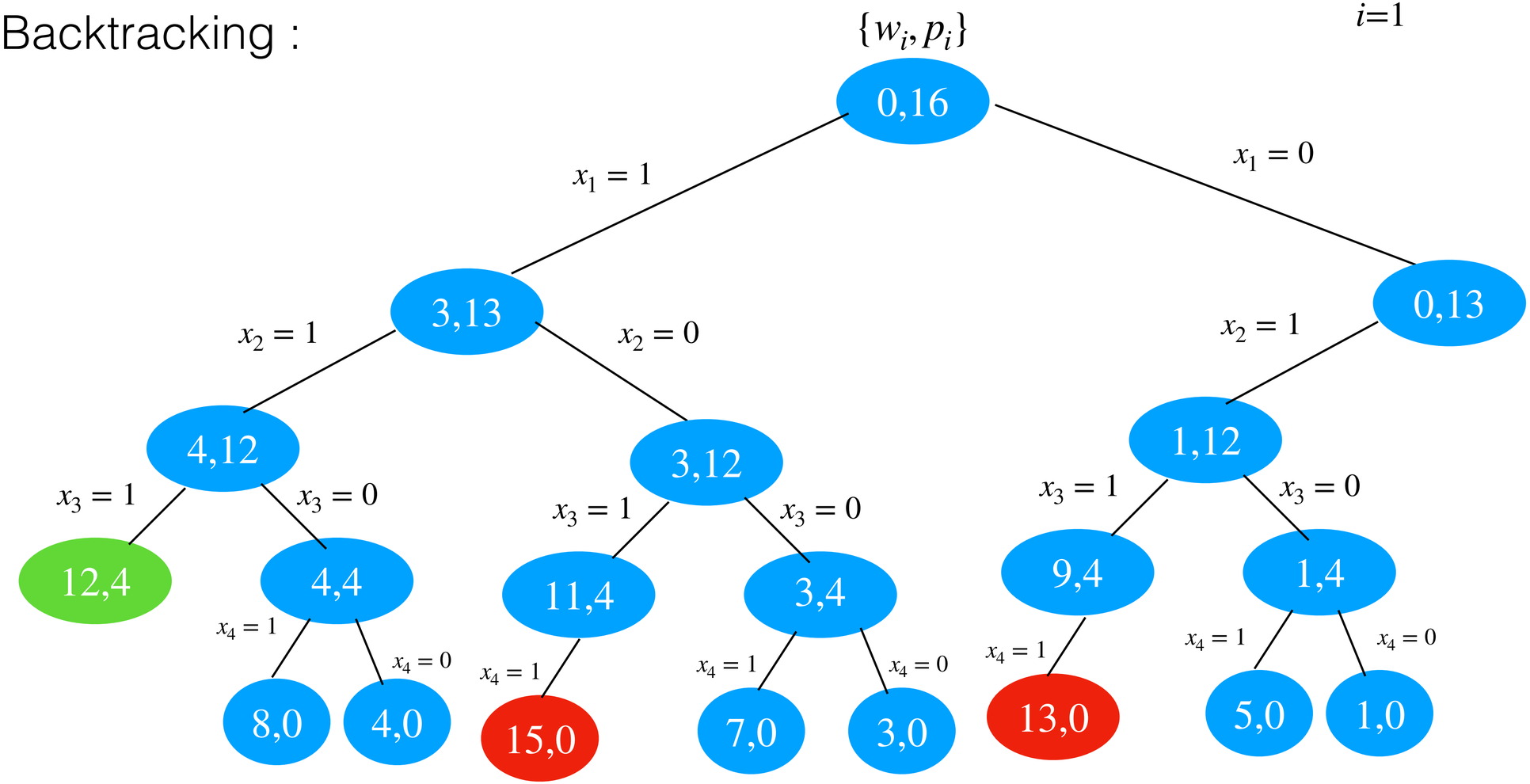
Backtracking :



Sum of Subsets Backtracking

Given the following set of items. $S = \{3,1,8,4\}$ find all its subsets s where $\sum_{i=1}^{n} s_i = m;\ \ m = 12$

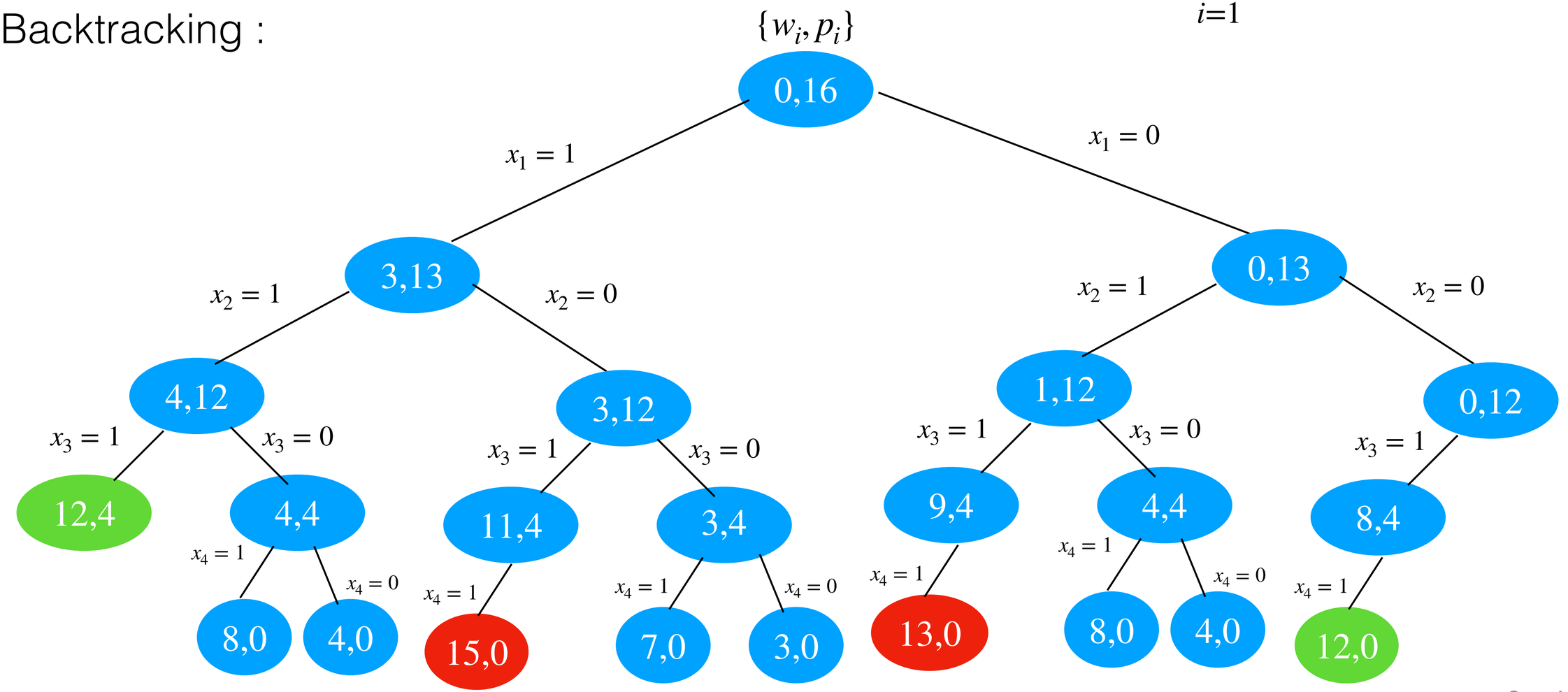
Backtracking :



Sum of Subsets Backtracking

Given the following set of items. $S = \{3,1,8,4\}$ find all its subsets s where $\sum_{i=1}^{n} s_i = m;\ \ m = 12$

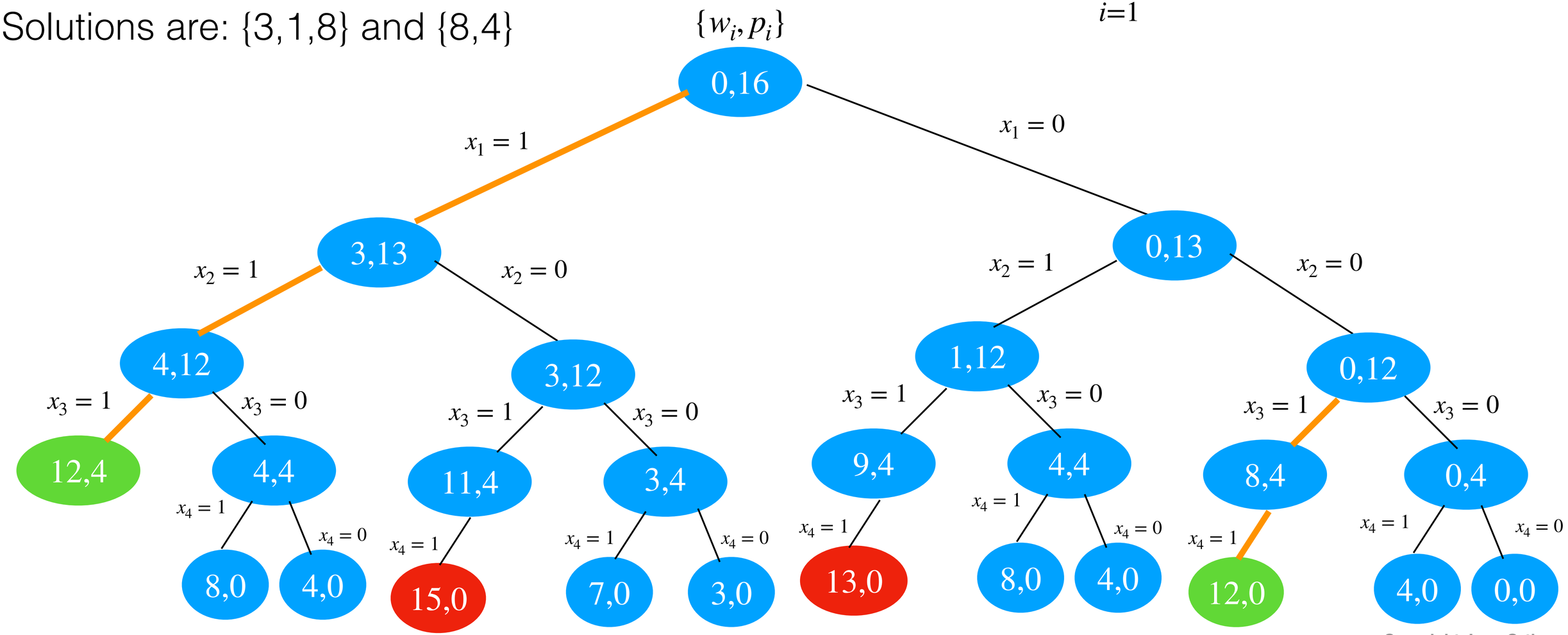
Backtracking :



Sum of Subsets Backtracking

Given the following set of items. $S = \{3,1,8,4\}$ find all its subsets s where $\sum_{i=1}^{n} s_i = m;\ \ m = 12$

Solutions are: {3,1,8} and {8,4}



Sum Of Subsets Pseudocode

Algorithm 1: Sum of Subsets Backtracking (All Solutions)

```
Function SumOfSubsets(Set, N, MaxSum):
    if N < 0 then
        return                                   ▷ exit from recursion
    Init: Node, NextNode, NodeIndex, Subset of size N, TmpSum
    Node ← Set[element at index N-1]
    if Node = MaxSum then
        Print: Node                              ▷ solution found!
    Append: Node to Subset
    NodeIndex ← (N − 2)
    while NodeIndex ≠ 0 do
        NextNode ← Set[element at index NodeIndex]
        Append: NextNode to Subset               ▷ included
        TmpSum ← Sum: All elements in Subset
        if TmpSum ≥ MaxSum then
            if TmpSum = MaxSum then
                Print: Subset                    ▷ solution found!
            Pop: Last element from Subset        ▷ not included
        Decrement: NodeIndex by 1
    end while
    SumOfSubsets(Set, N-1, MaxSum)   ▷ recursive call to itself
```

Backtracking N-Queens

Backtracking N-Queens

- Problem statement:
  - Given a $N * N$ board, place $N - Queens$ such that no queens attack each other. Attacks occur when queens are placed in any of the following ways:
    - Same column
    - Same row
    - Diagonal

Backtracking N-Queens

For example, we are given a $4x4$ board, and $Q = \{q_1, q_2, q_3, q_4\}$ representing $4 - Queens$:

Diagonal

	1	2	3	4
1	q_1			
2		q_2		
3			q_3	
4				q_4

Same column

	1	2	3	4
1	q_1			
2	q_2			
3		q_3		
4				q_4

Same row

	1	2	3	4
1	q_1	q_2		
2				
3			q_3	
4				q_4

	1	2	3	4
1		q_1		
2				q_2
3	q_3			
4			q_4	

	1	2	3	4
1			q_1	
2	q_2			
3				q_3
4		q_4		

Brute Force Approach N-Queens

A brute force approach would consider all the steps to find all the possible solutions

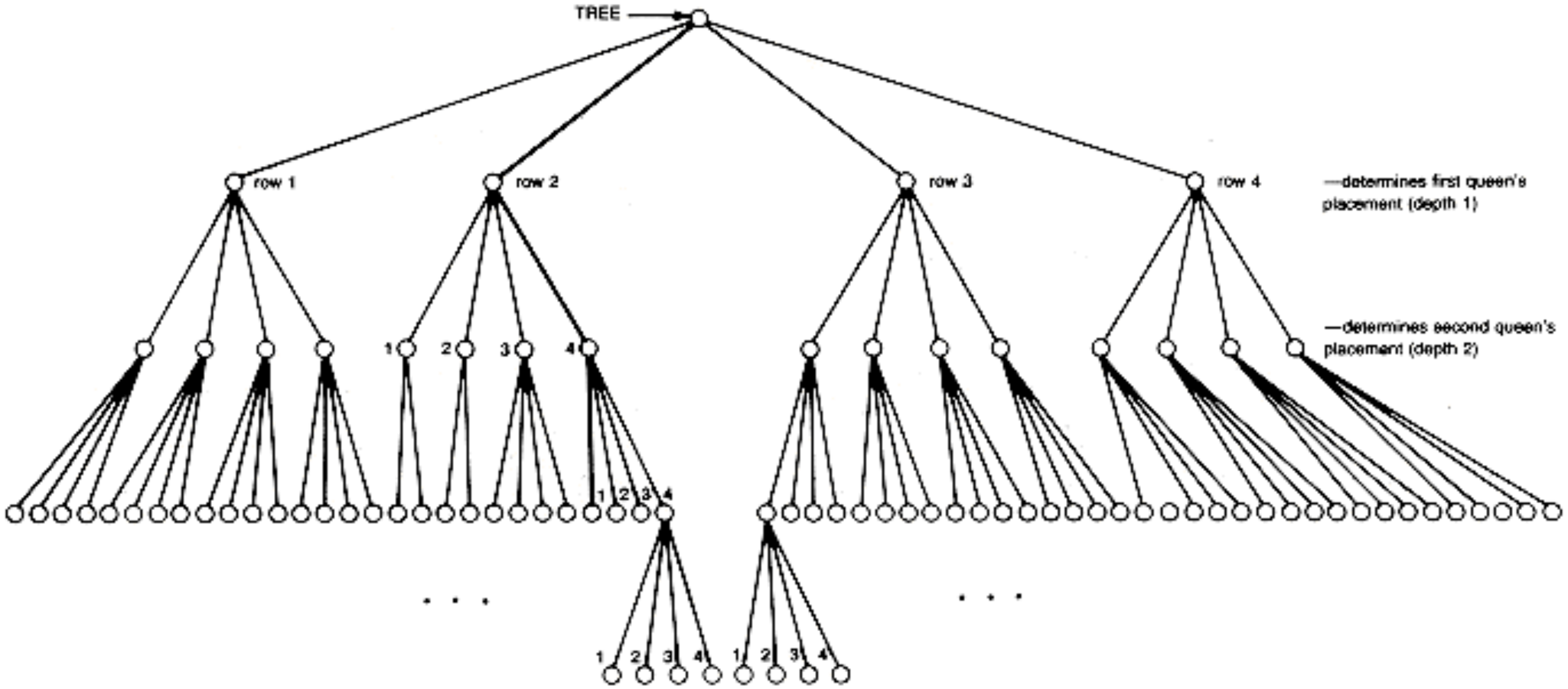
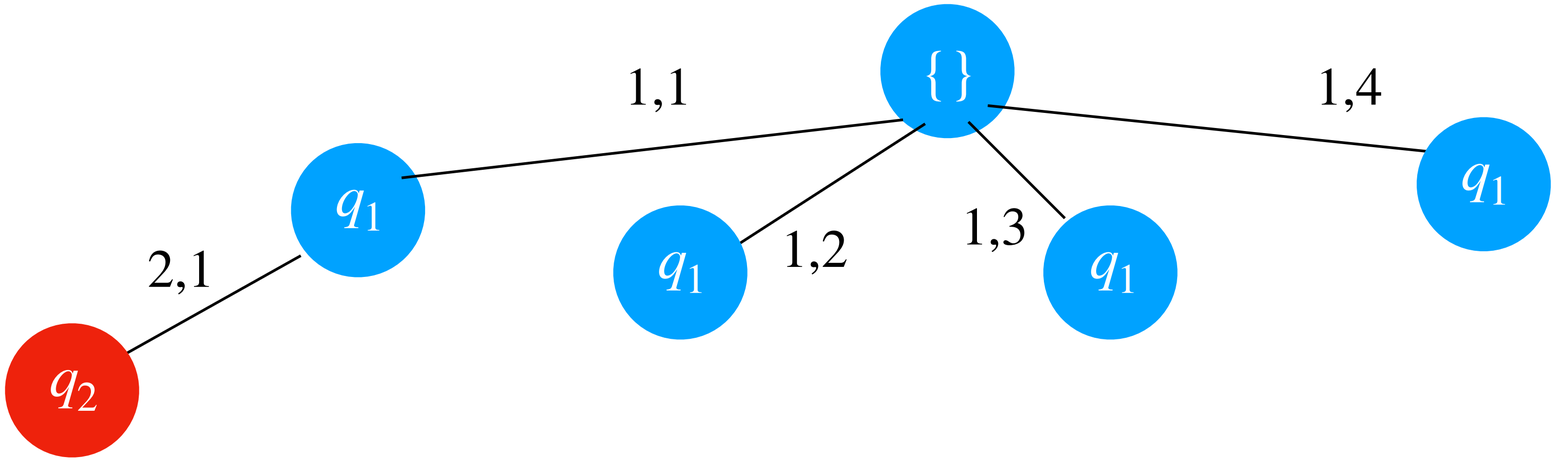


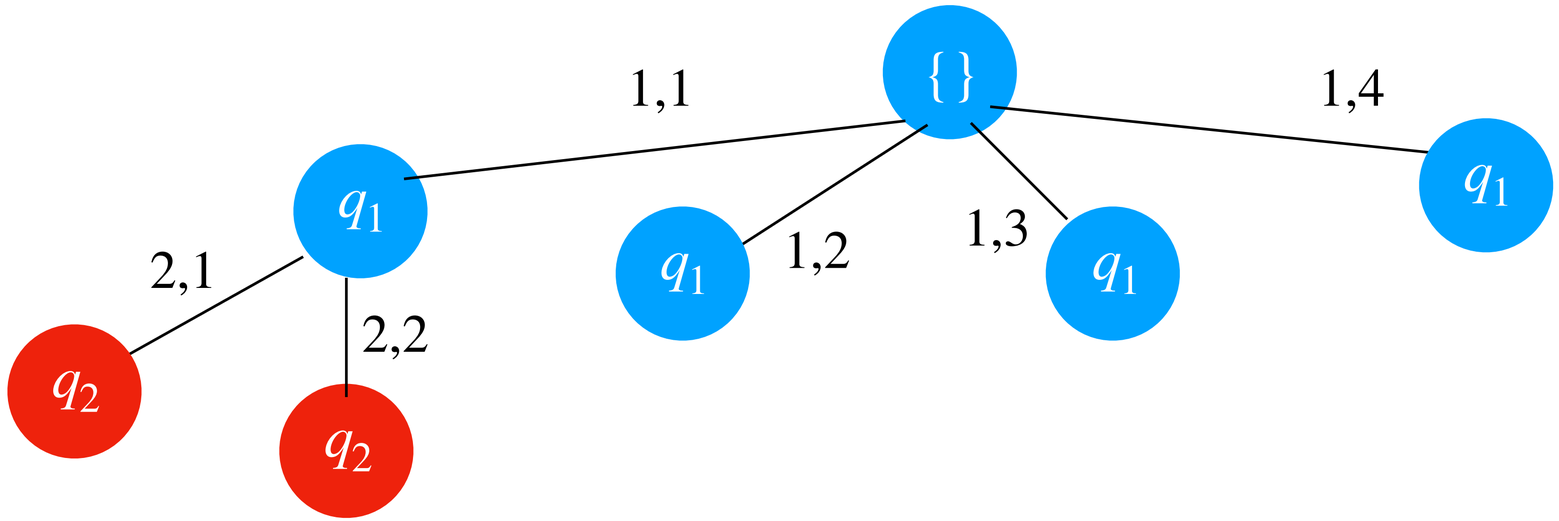
Image Source: orion.com

Backtracking N-Queens

Links Numbers Repesent $\{row, column\}$



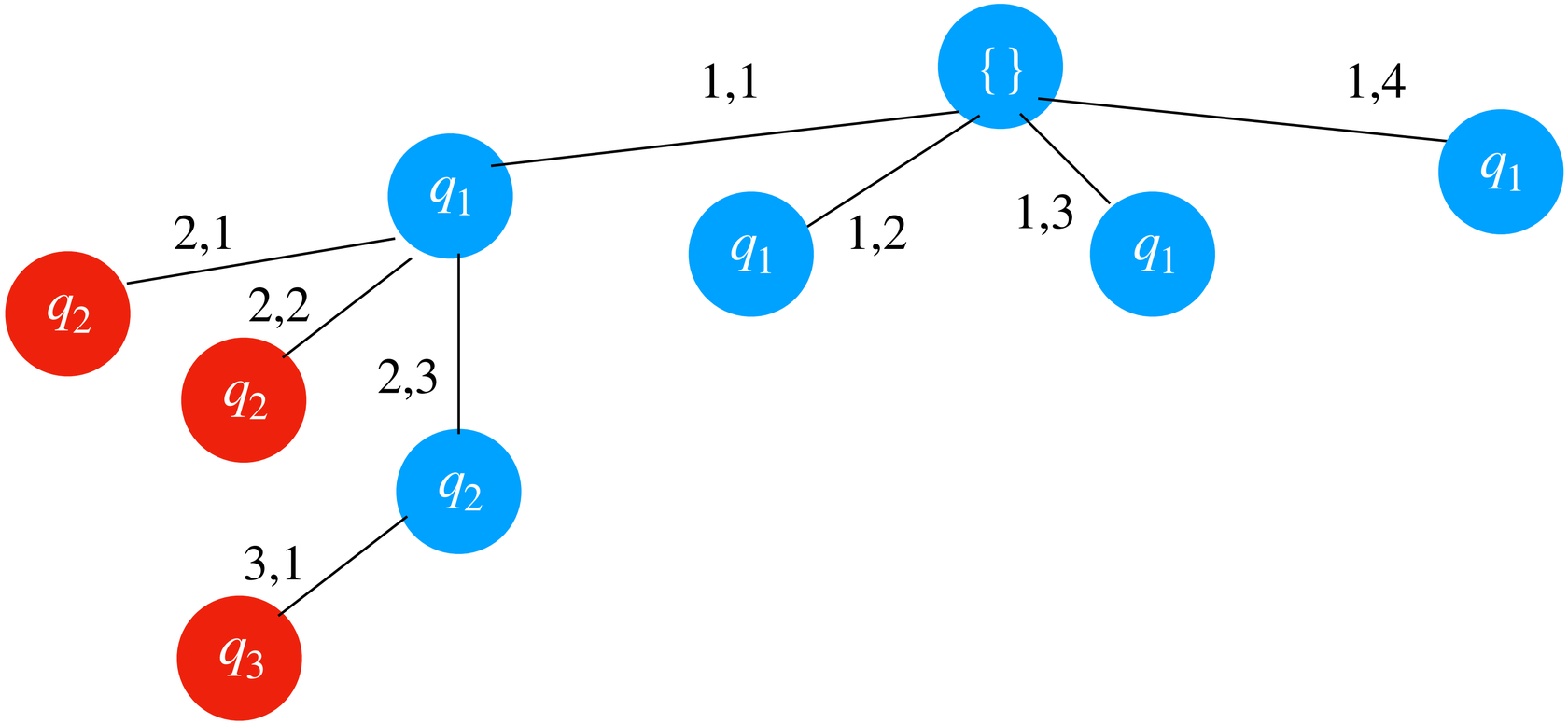
	1	2	3	4
1	q_1			
2	q_2			
3				
4				



	1	2	3	4
1	q_1			
2		q_2		
3				
4				

Backtracking N-Queens

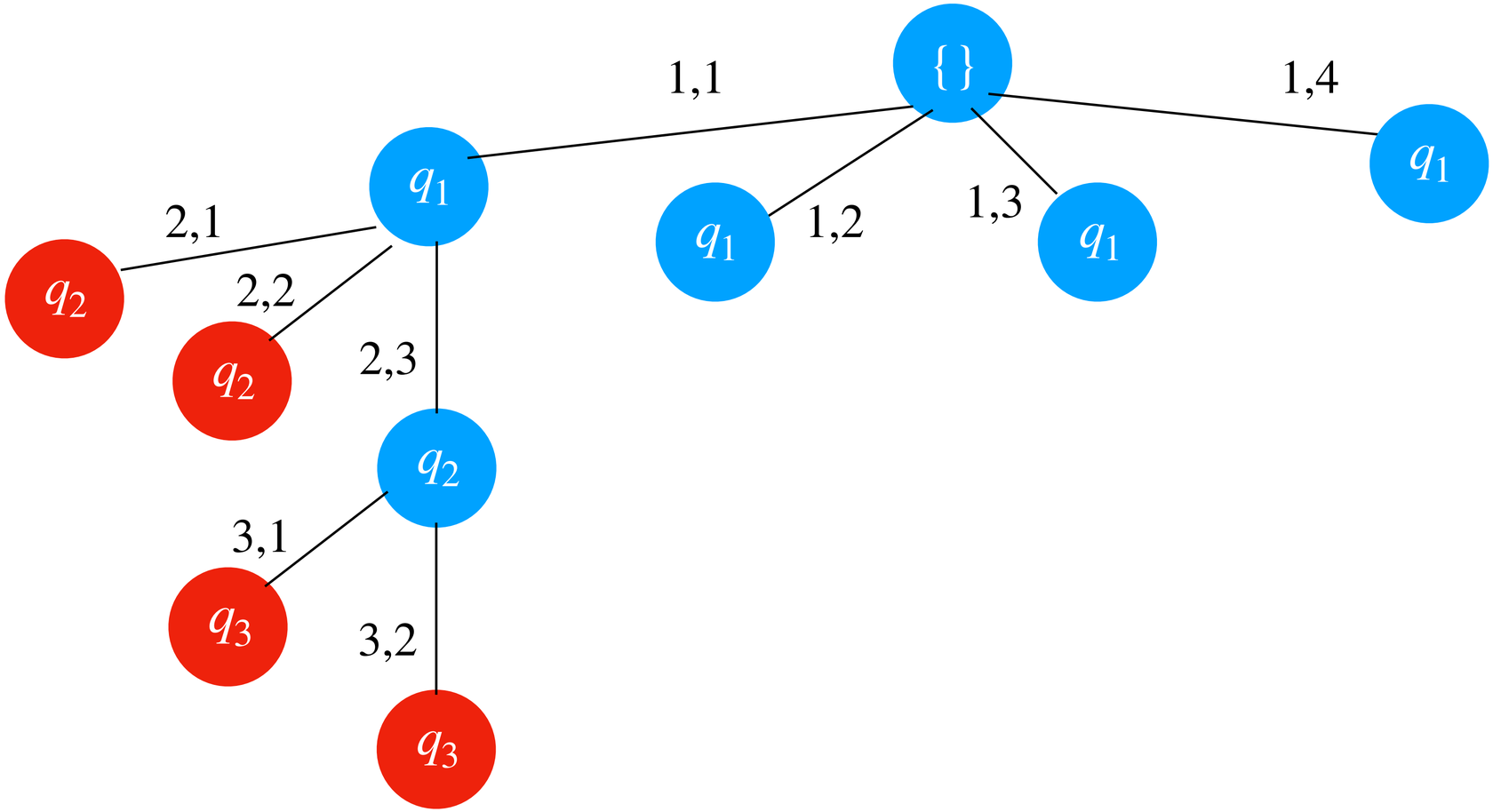
Links Numbers Repesent $\{row, column\}$



	1	2	3	4
1	q_1			
2			q_2	
3	q_3			
4				

Backtracking N-Queens

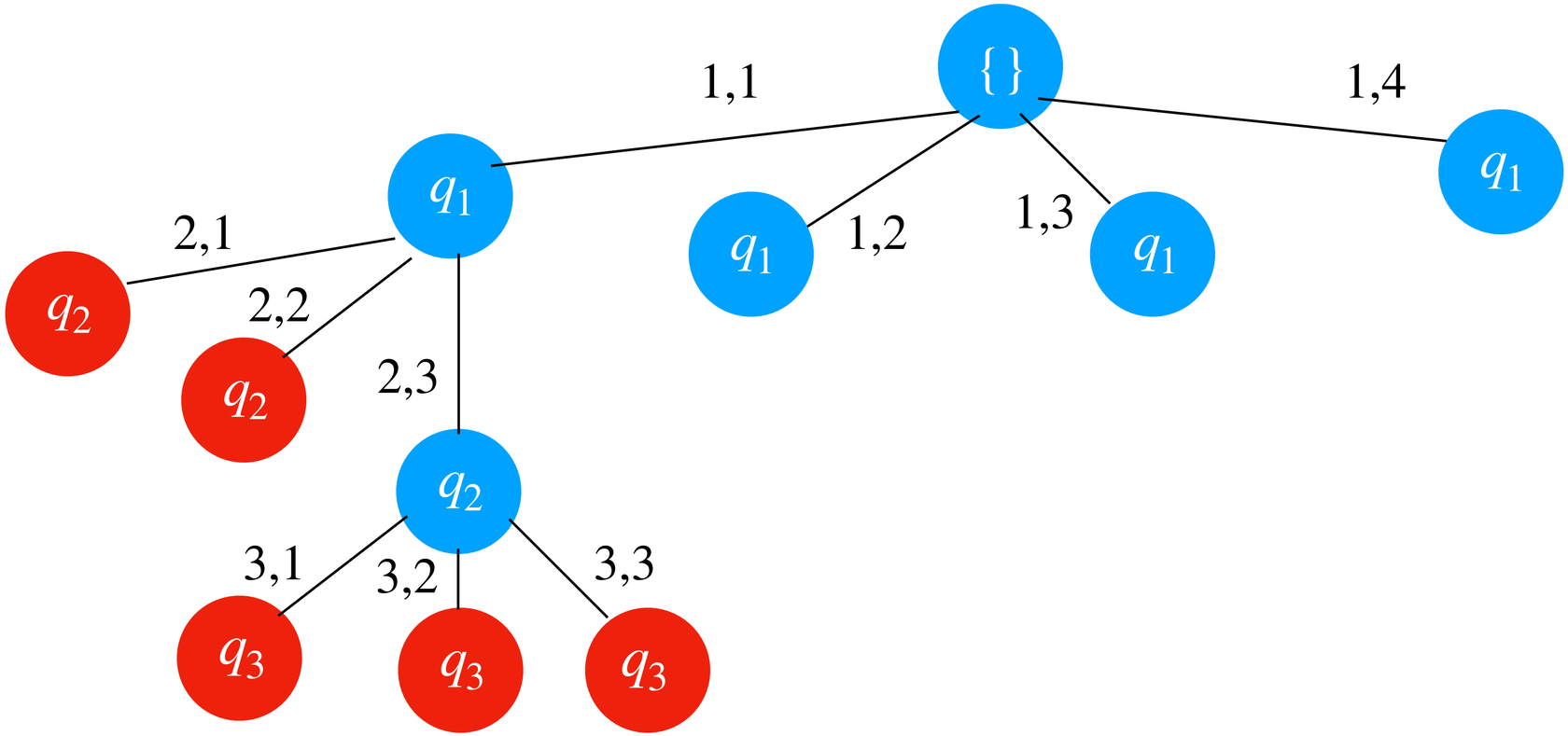
Links Numbers Repesent $\{row, column\}$



	1	2	3	4
1	q_1			
2			q_2	
3		q_3		
4				

Backtracking N-Queens

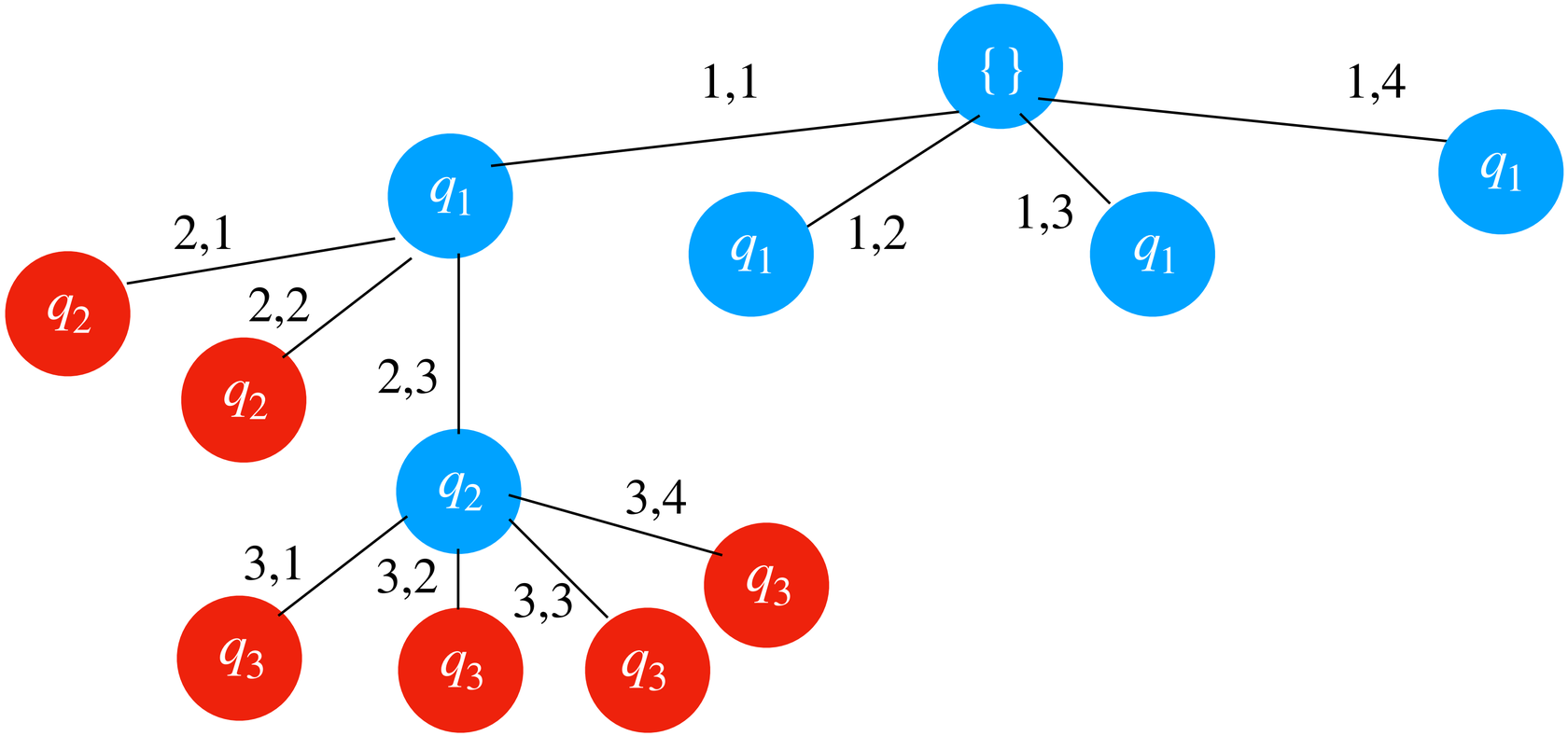
Links Numbers Repesent $\{row, column\}$



	1	2	3	4
1	q_1			
2			q_2	
3			q_3	
4				

Backtracking N-Queens

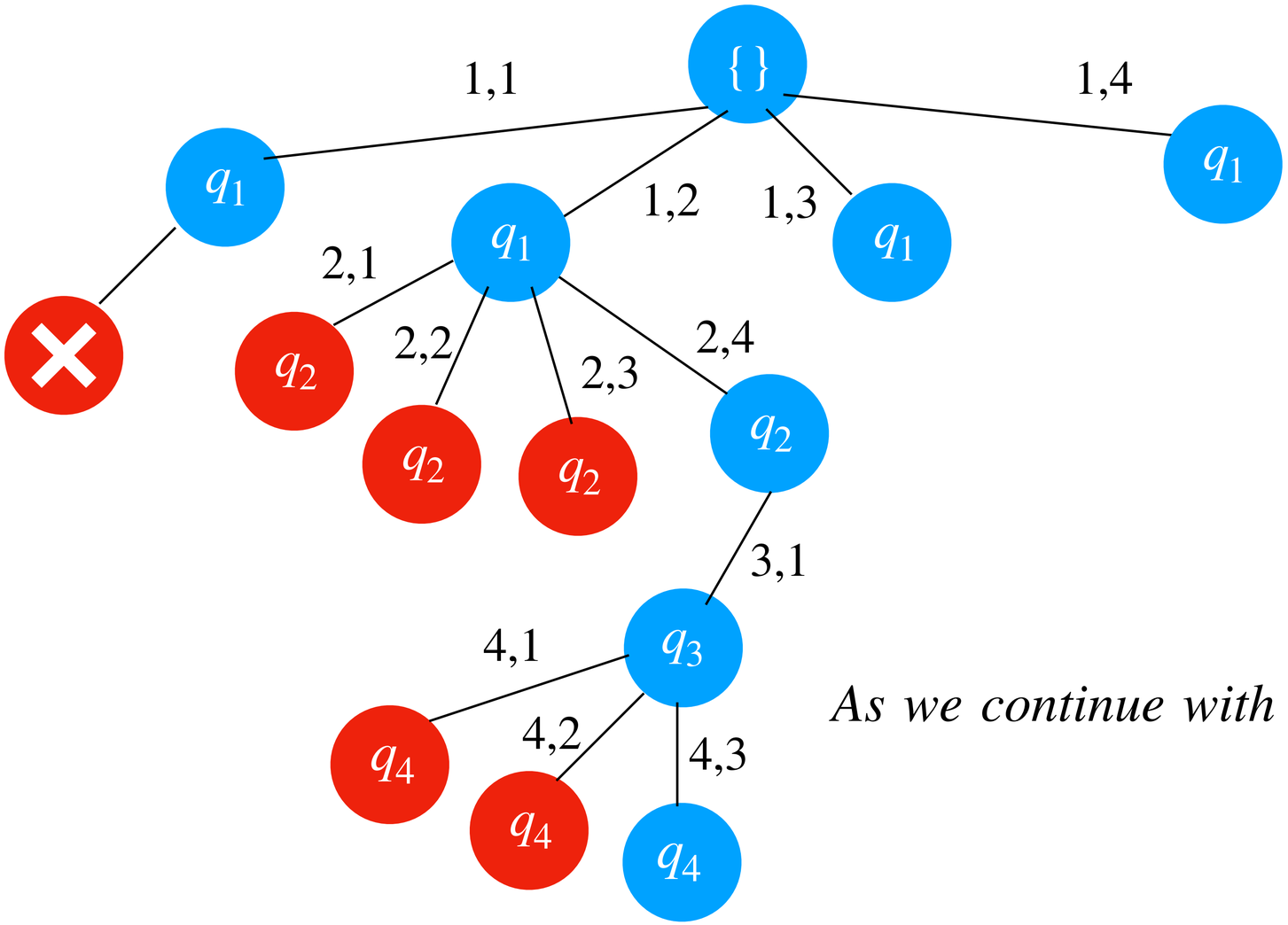
Links Numbers Repesent $\{row, column\}$



	1	2	3	4
1	q_1			
2			q_2	
3				q_3
4				

Backtracking N-Queens

After a few more iterations . . .



Solution Found!

	1	2	3	4
1		q_1		
2				q_2
3	q_3			
4			q_4	

As we continue with the process, more solutions will be found

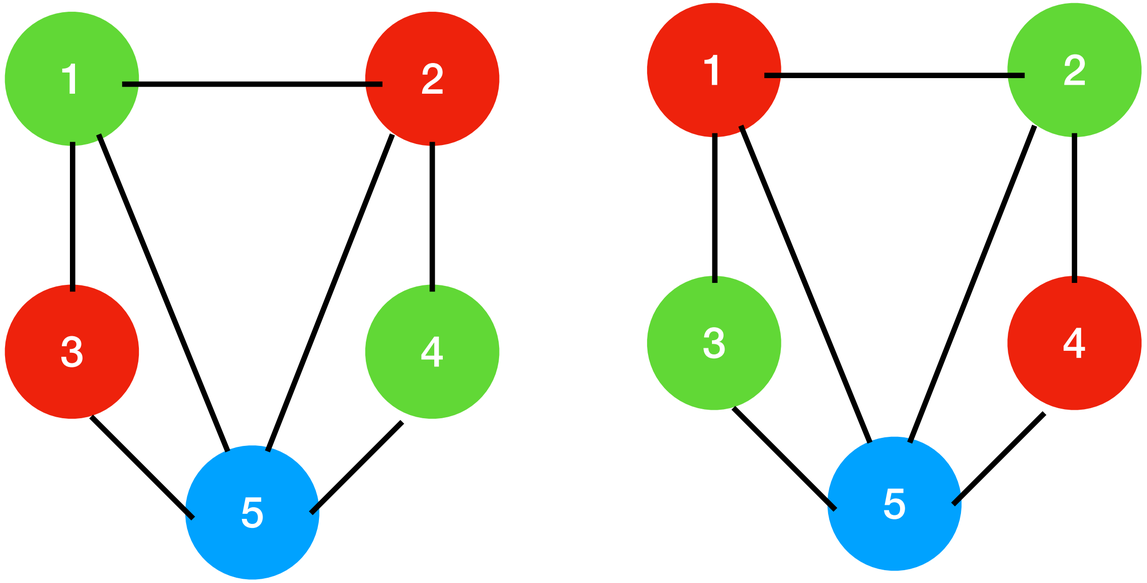
Backtracking N-Queens
Time Complexity Analysis

- Brute Force:
  - Solving this problem by brute force implies that we need to examine all the possible permutations. For instance, $\{0,\ldots\ldots\ldots, (n-1)\}\ is\ O(n!)$
- Backtracking:
  - One solution: the best case occurs if you find the solution before exploiting all the possible arrangements (rarely happens). Also the time complexity of finding the first solution depends on your implementation
  - All the solutions: the same as brute force $O(n!)$. However, in average time complexity hits the lower bounds of $O(n!)$ since there are many permutations that are abandoned during the process.
  - The backtracking algorithm can be further improved using memoization for all the recursive calls.

Backtracking Graph Coloring

M-Graph Coloring Backtracking

Given an undirected graph $G = \{v_1, v_2, \ldots, v_n\}$, and a set of colors $M = \{c_1, c_2, \ldots c_m\}$, color all the vertices of the graph such that no two adjacent vertices share the same color.



There are many possible solutions for

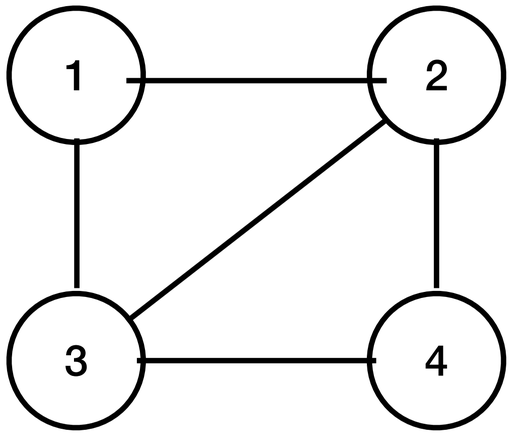
$G = \{1,2,3,4,5\}$

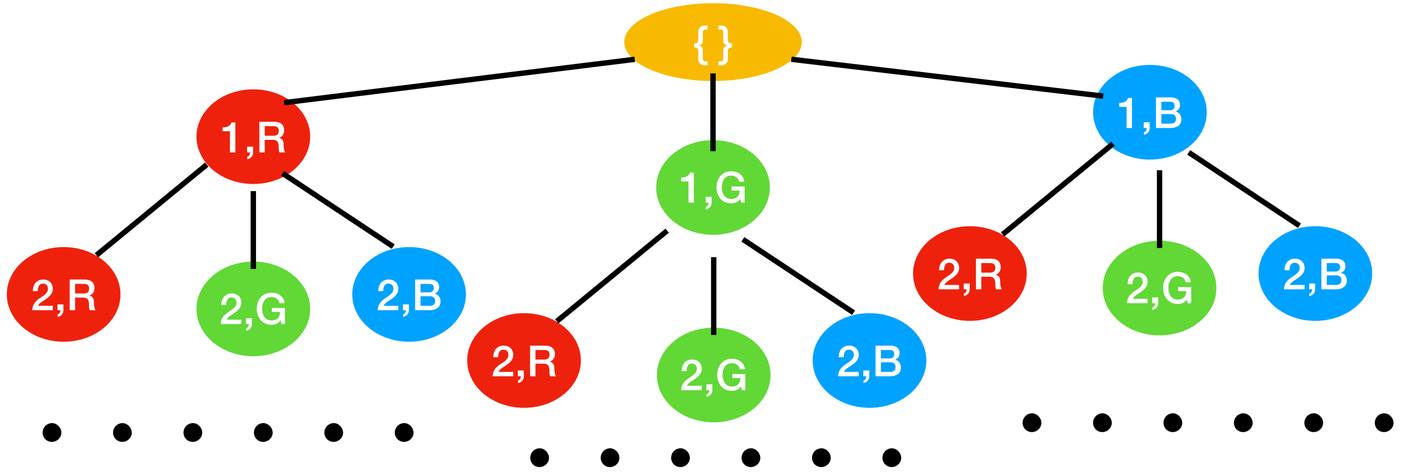
$M = \{red, green, blue\}$

Note that two vertices are adjacent if they are connected by an edge, and two edges are adjacent is they share a vertex.

M-Graph Coloring Brute Force Approach

A brute force approach will compute all the possible ways to color a graph without bounding. That's it all the steps are considered.



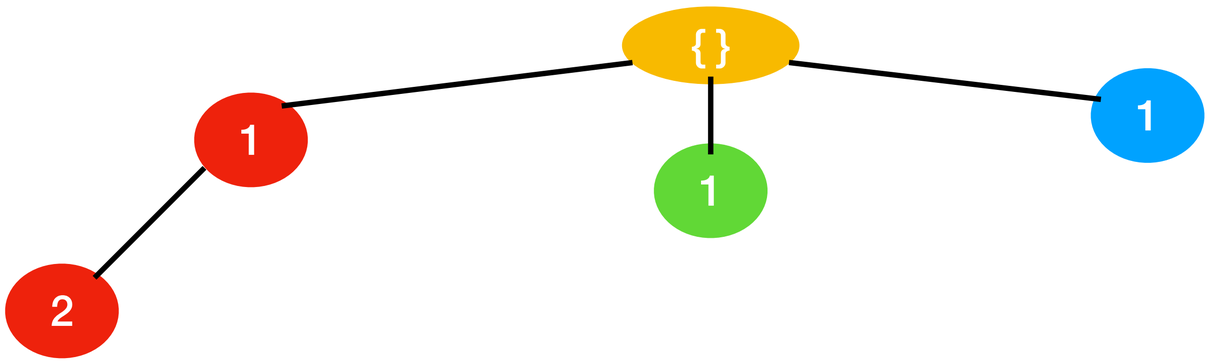


$$Level\ 1 = 3^0\ vertex$$
$$Level\ 1 = 3^1\ vertices$$
$$Level\ 1 = 3^2\ vertices$$
$$Level\ 1 = 3^3\ vertices$$
$$Level\ 1 = 3^4\ vertices$$

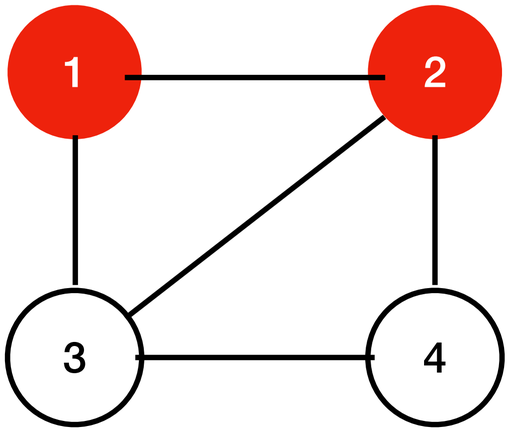
$$\sum_{i=1}^{n} 3^4 = \frac{3^{4+1} - 1}{2}$$

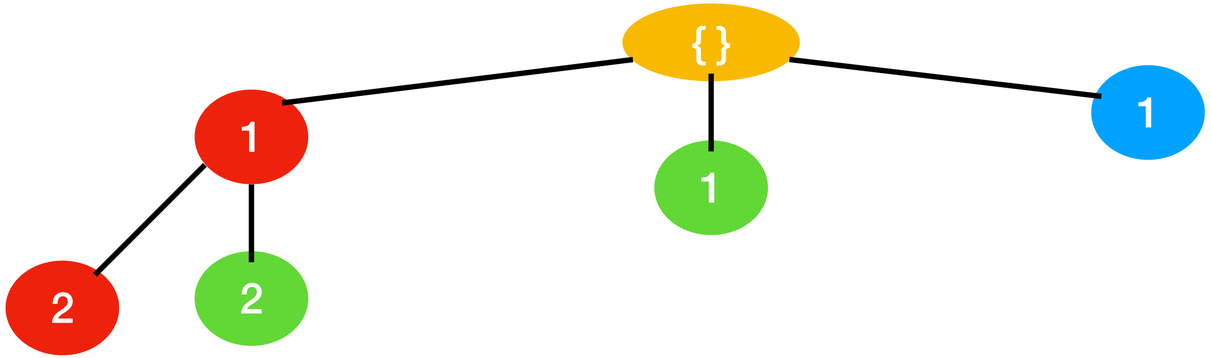
$$O(3^{n+1}) = O(m^{n+1})$$

M-Graph Coloring Backtracking

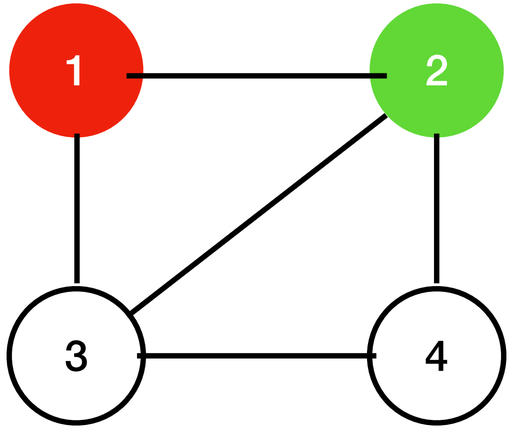


No!

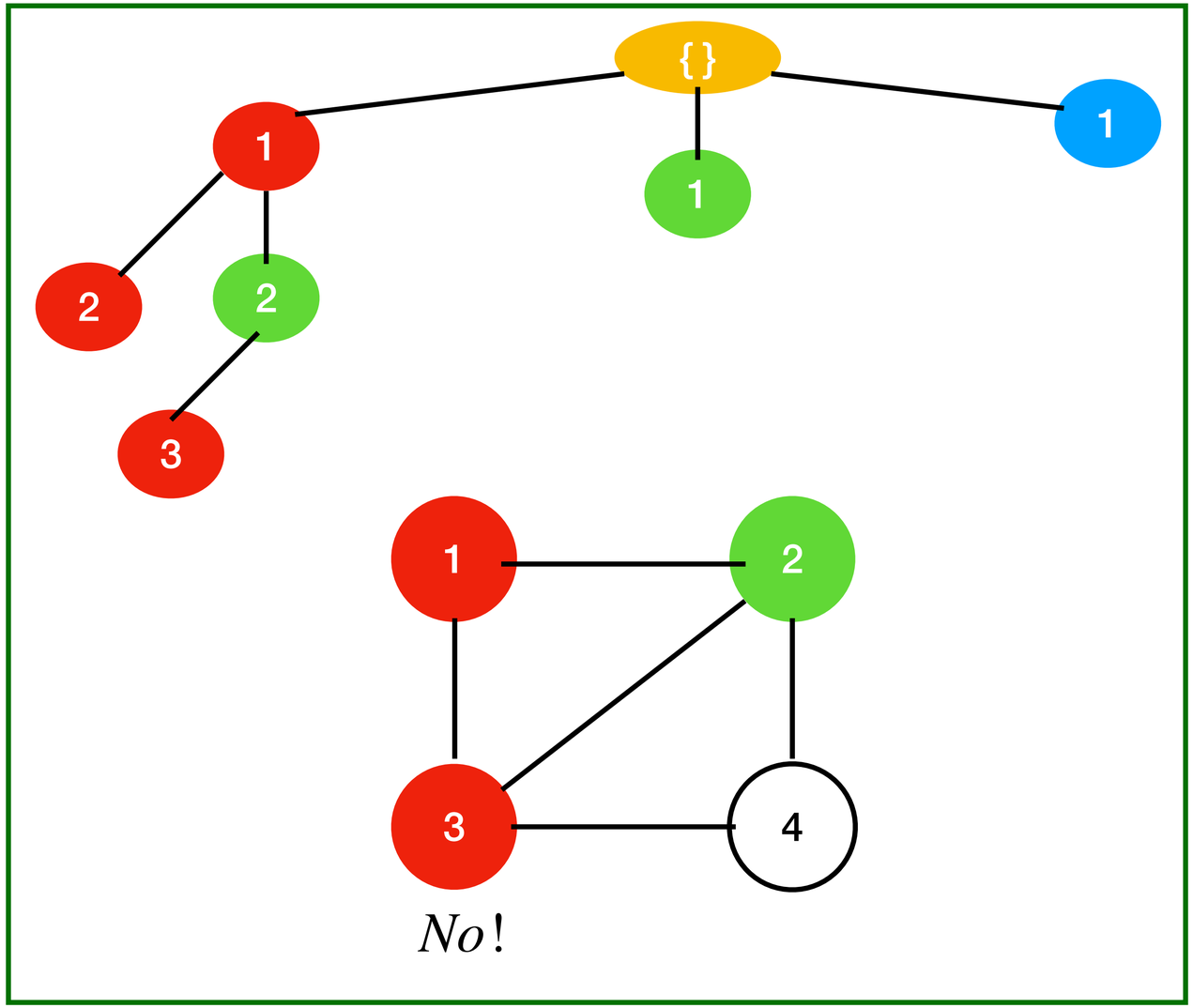


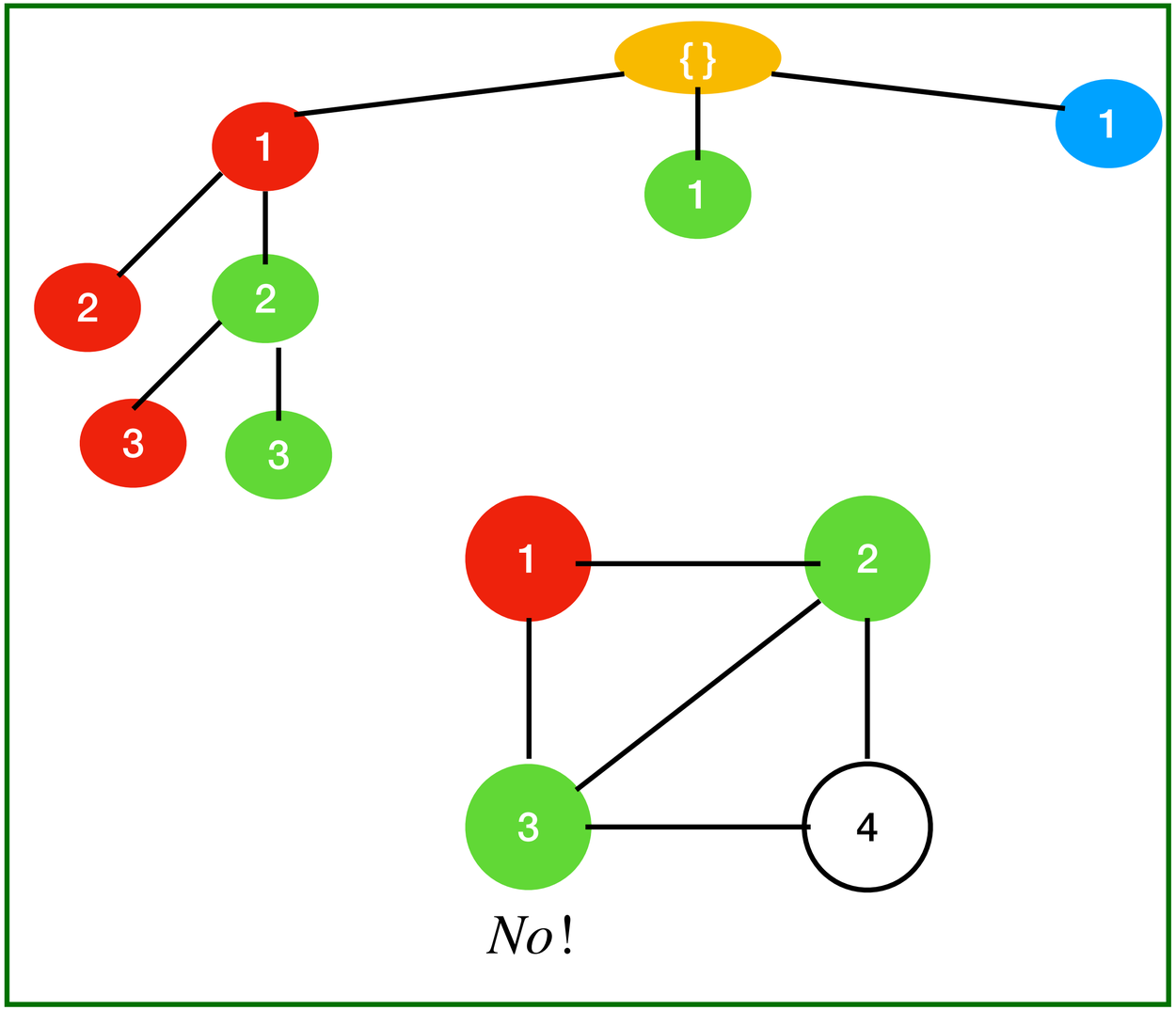


Yes!

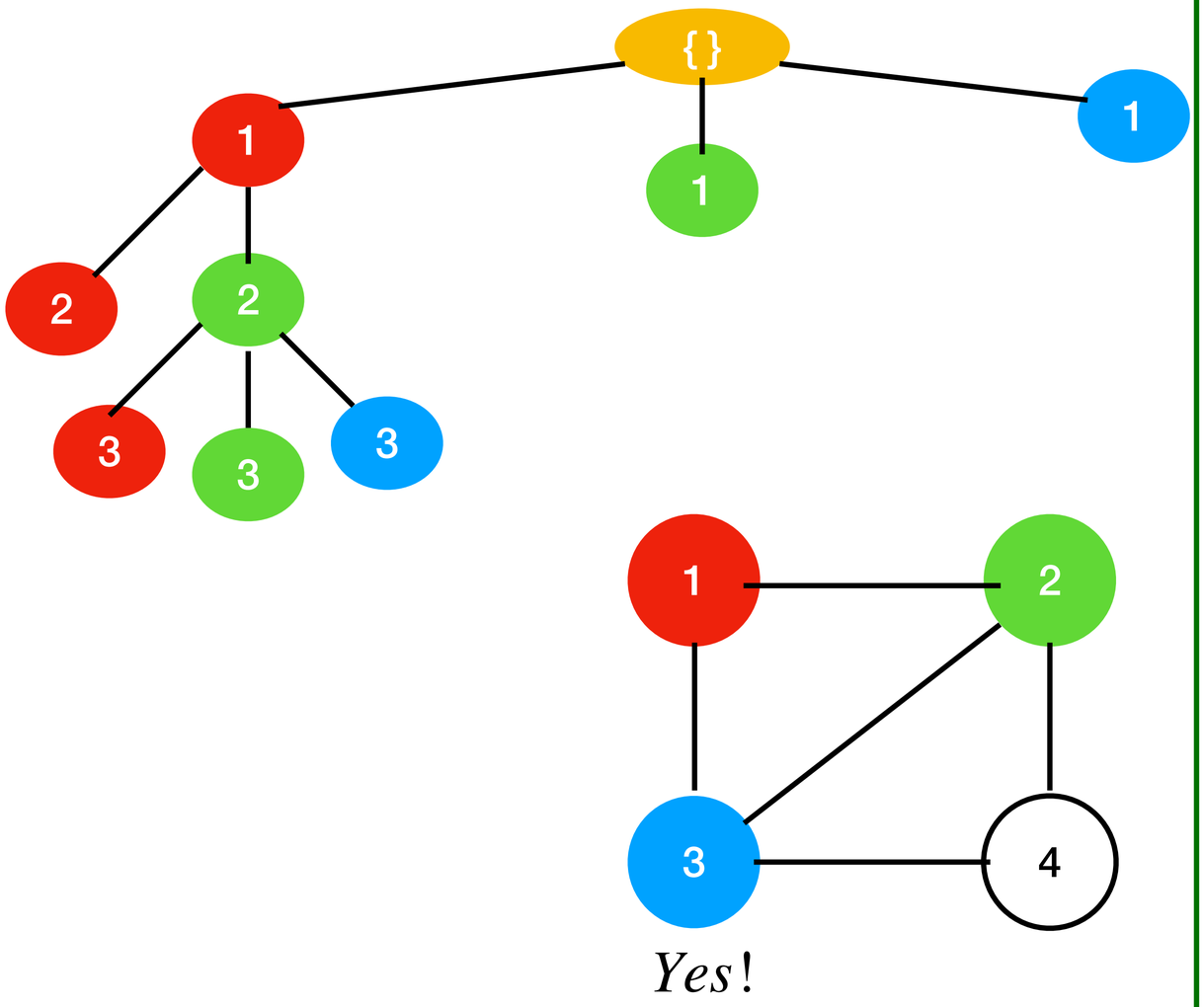


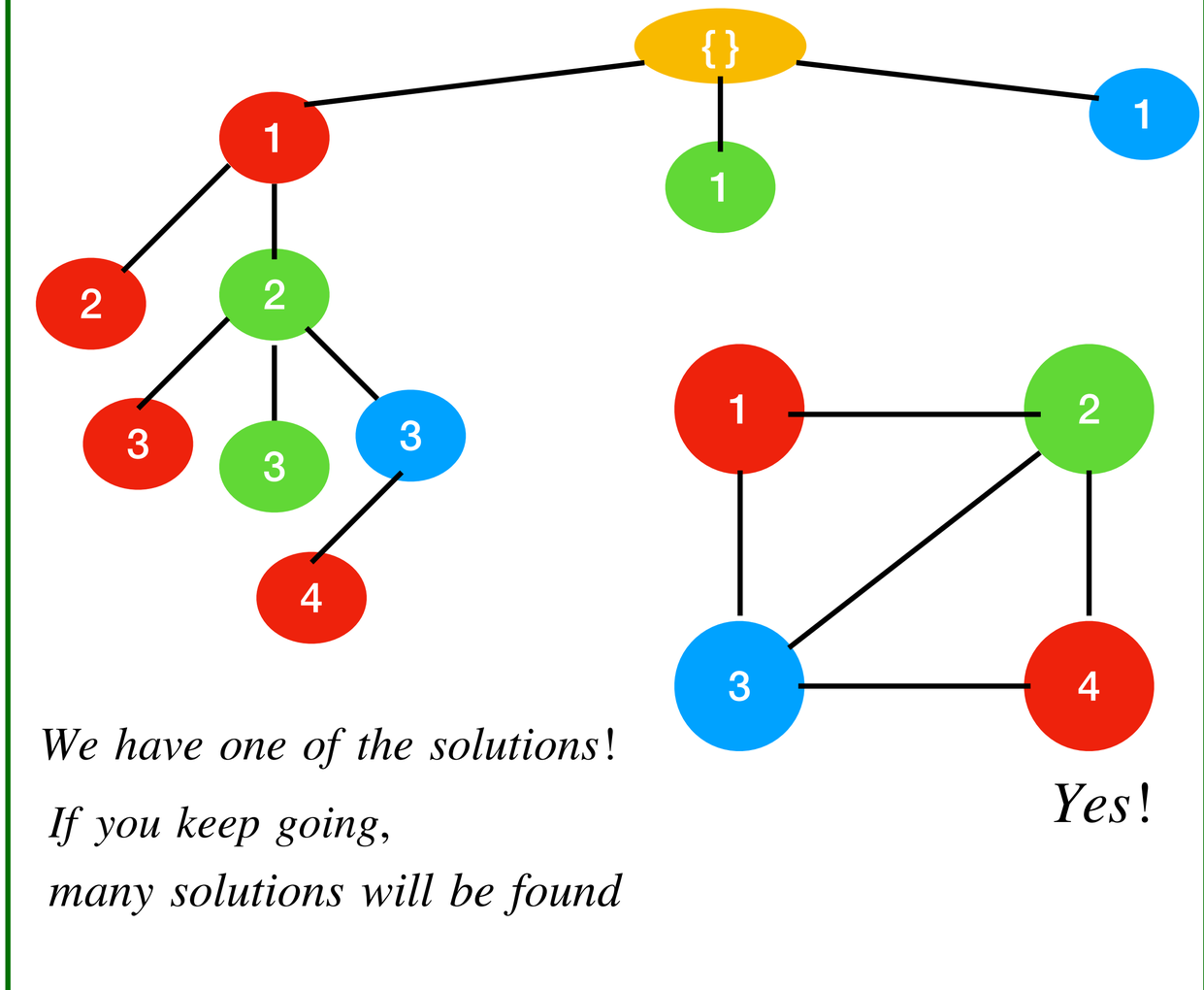
M-Graph Coloring Backtracking





M-Graph Coloring Backtracking



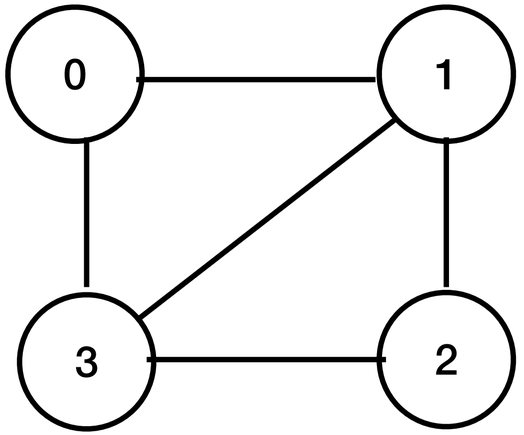


M-Graph Coloring Backtracking Pseudocode

Algorithm 1: Graph Coloring Algorithm

```
Function GraphColoring(Graph, Colors, ColorNumbers, Row):
    Init: Colored (array of size N), Color
    if (ColorNumber + 1) = N then
        Print: Colored                          ▷ exit from recursion
    Color ← 1                                   ▷ first color
    foreach Color in the range of ColorNumbers+1 do
        if isSafe(Graph, Row, Color, Colored) then
            Colored[at index Row] ← Color
            GraphColoring(Graph, Colors, ColorNumbers, Row+1)
        else
            Colored[index at Row] ← 0           ▷ no solution found
        end if
    end foreach

Function IsSafe(Graph, Row, Color, Colored):
    foreach Column in Graph do
        Connected ← Graph[Row][Column]          ▷ is 1 or 0
        if Vertex is Connected and Color in Colored[Column] then
            return False
    end foreach
    return True
```



G	0	1	2	3
0	1	1	0	1
1	1	1	1	1
2	0	1	1	1
3	1	1	1	1

0 = *not connected*
1 = *connected*

Backtracking M-Graph Coloring Time Complexity Analysis

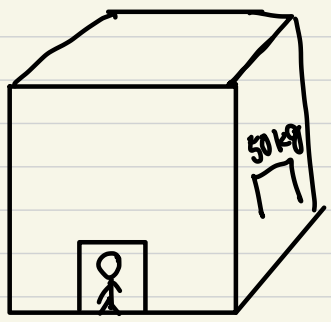
- Brute Force:
  - Solving this problem by brute force implies that we need to color each node visited m times. Therefore, time complexity using a brute force approach is $O(m^{n+1})$ where m is the total number of colors, and n the total number of nodes to be colored.
- Backtracking:
  - The recursion takes n times since it needs to visit all the n nodes recursively, for every visited node, the function needs to check is there is a valid color for that node by visiting all its adjacent nodes. Therefore, the time complexity of solving this problem with backtracking is $O(nm^n)$, which is slightly better than the one from the brute force approach

Branch : DFS, Leastcost, BFS

Bound : any this week

Knapsack problem

BestBuy



Stands with products

Items = { 1, 2, 3, 4 }

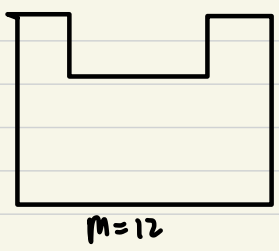
Weights = { 5, 3, 7, 2 }

Profit = { 12, 5, 10, 7 }

$12 + $5

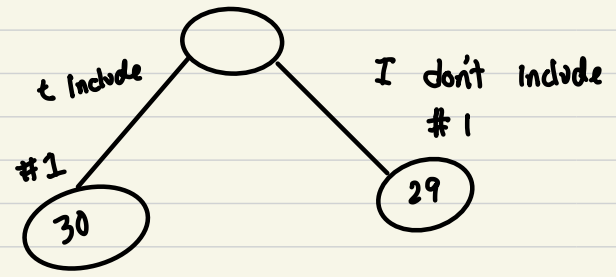
12 - 5 = 7

7 - 3 = 4



Backtracking Branch

Branch:



cost = $12 + $5 + $10/7kg (4kg)

= $22.71 = -22.71

bounds = $12 + $5 = $17

	1	2	3	4
Wi	0	3	+ 7	+ 2
Pi	0	+ $5	+ $10	+ $7

Cost = $5 + $10 + $7 = -22

bound = -22

$X_1 = 1$ $X_1 = 0$

$c < c$

$b > b$

	1	2	3	4
Wi	0	3	+ 0	+ 2
Pi	0	+ $5	+ 0	+ $7

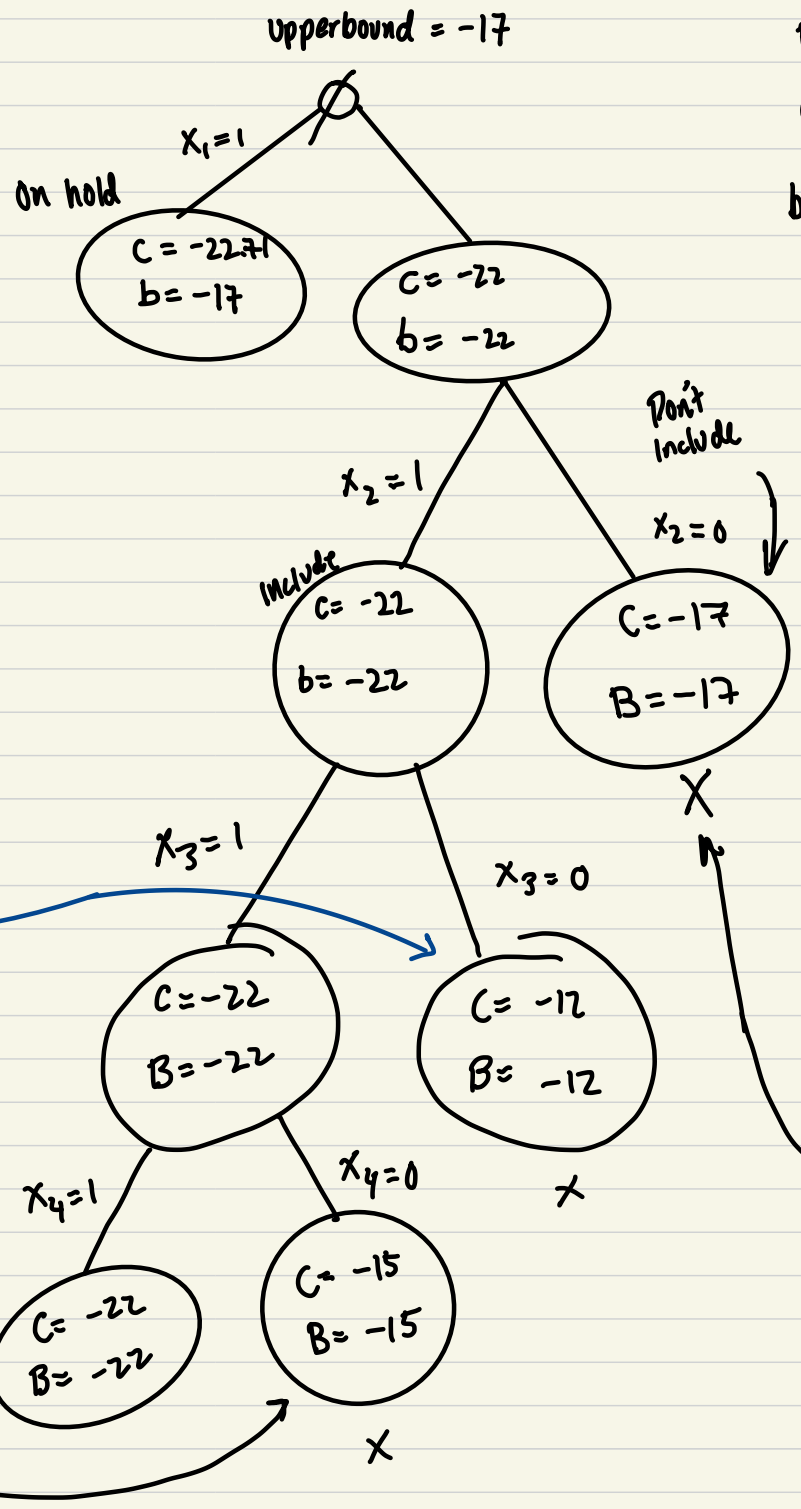
Cost = 5+7 = -12

Bound = 5+7 = -12

	1	2	3	4
Wi	0	3	+ 7	+ 0
Pi	0	+ $5	+ $10	+ 0

Cost = -15

Bound = -15



Branch: Least cost

Bounds:

Cost: $\sum p_i$ + Fracc

bound: $\sum p_i$

$X_i = 1$ item i is included

$X_i = 0$ item i is not included

A ✓ B on hold

Cost A > Cost B

Bound A < Bound B

We take A

	1	2	3	4
Wi	0	0	+ 7	+ 2
Pi	0	0	+ $10	+ $7

Cost = 0 + 0 + 10 + 7 = -17

bound = 0 + 0 + 10 + 7 = -17

	1	2	3	4
w_i	5	0	7	0
p_i	12	0	$10	0

cost = -22

bound = -22

	1	2	3	4
w_i	5	0	0	2
p_i	12	0	0	$7

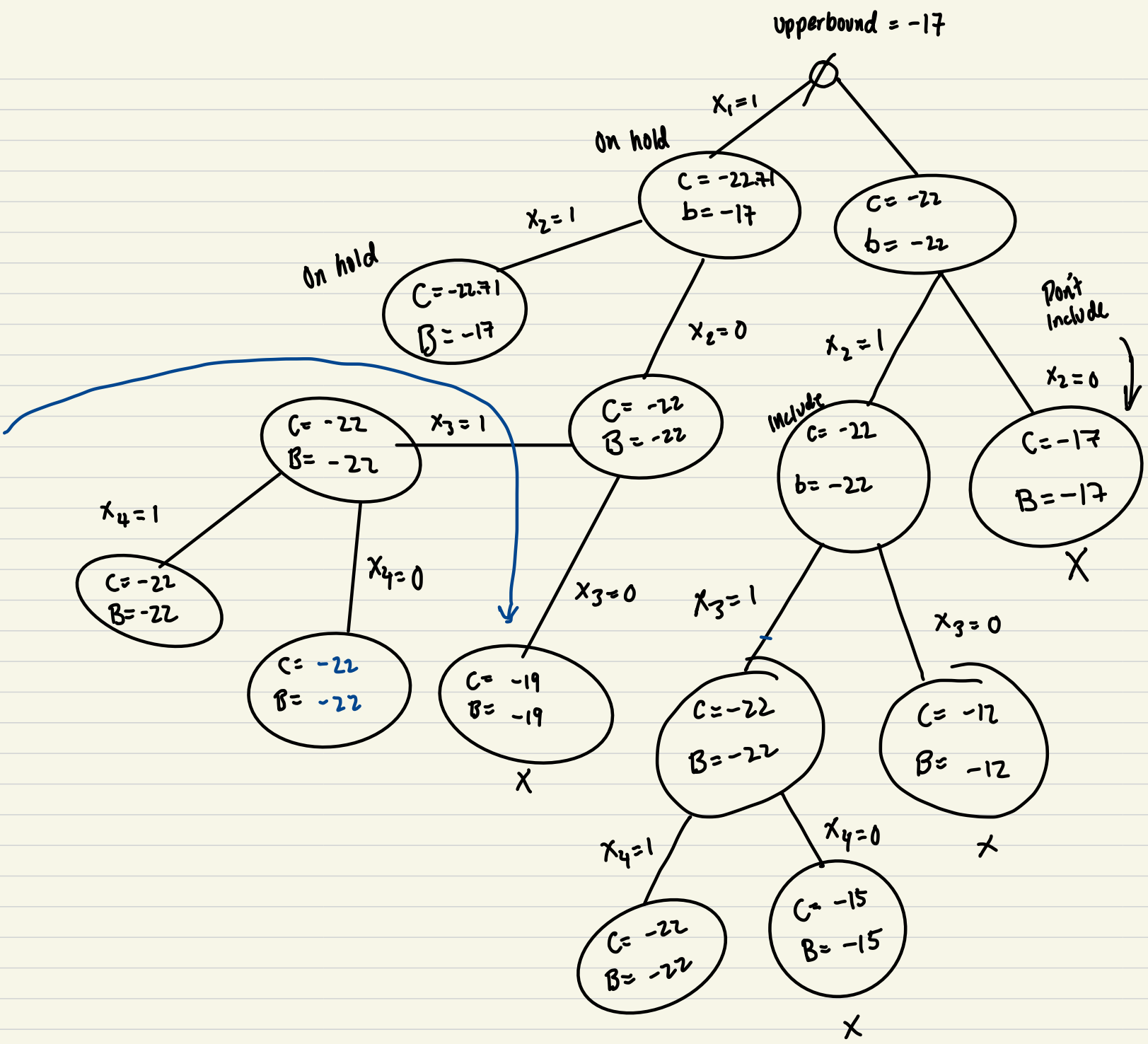
cost = -19

bound = -19

	1	2	3	4
w_i	5	0	7	0
p_i	12	0	$10	0

cost = -22

bound = -22



	1	2	3	4
w_i	5	3	0	2
p_i	12	5	0	7

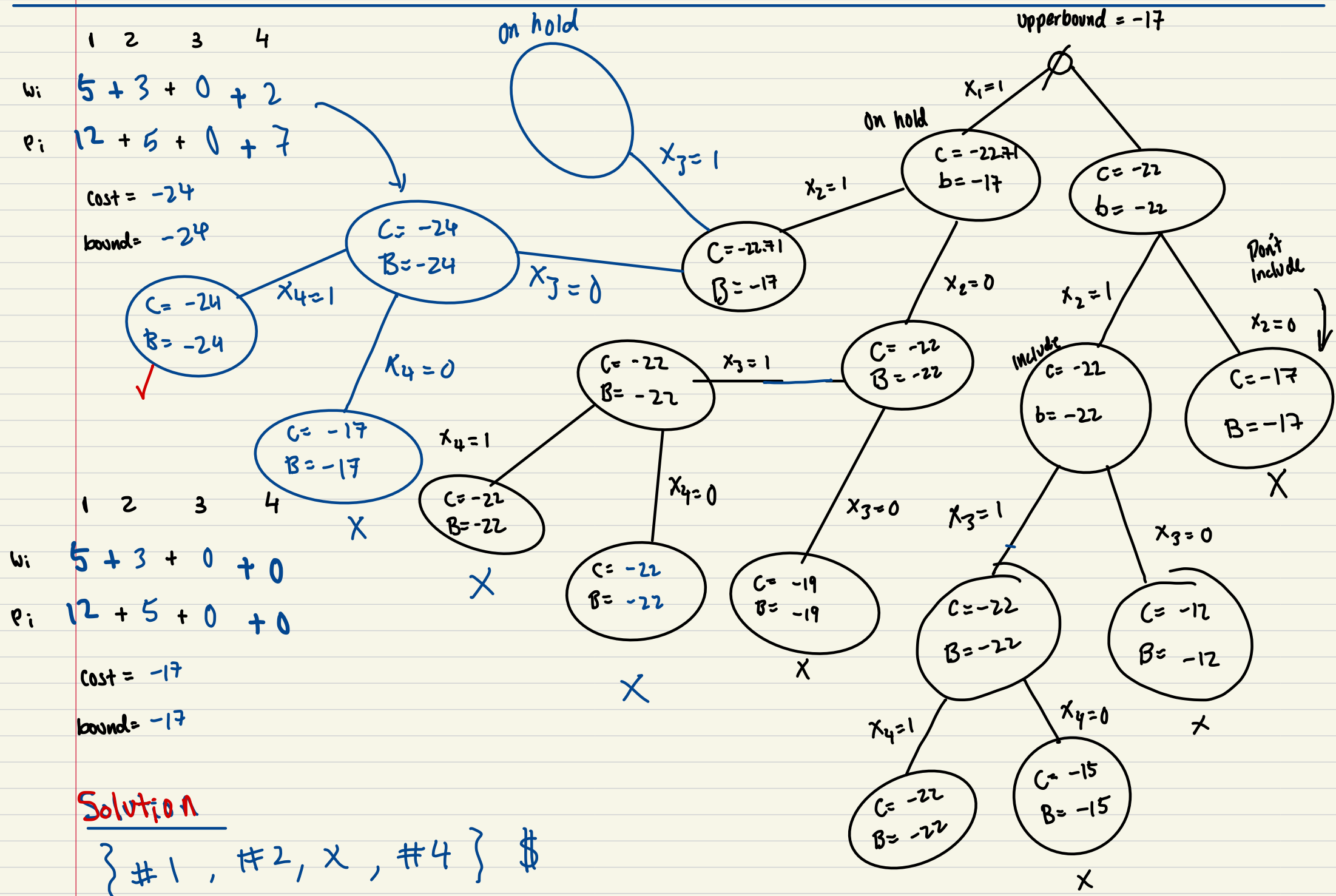
cost = -24

bound = -24

	1	2	3	4
w_i	5	3	0	0
p_i	12	5	0	0

cost = -17

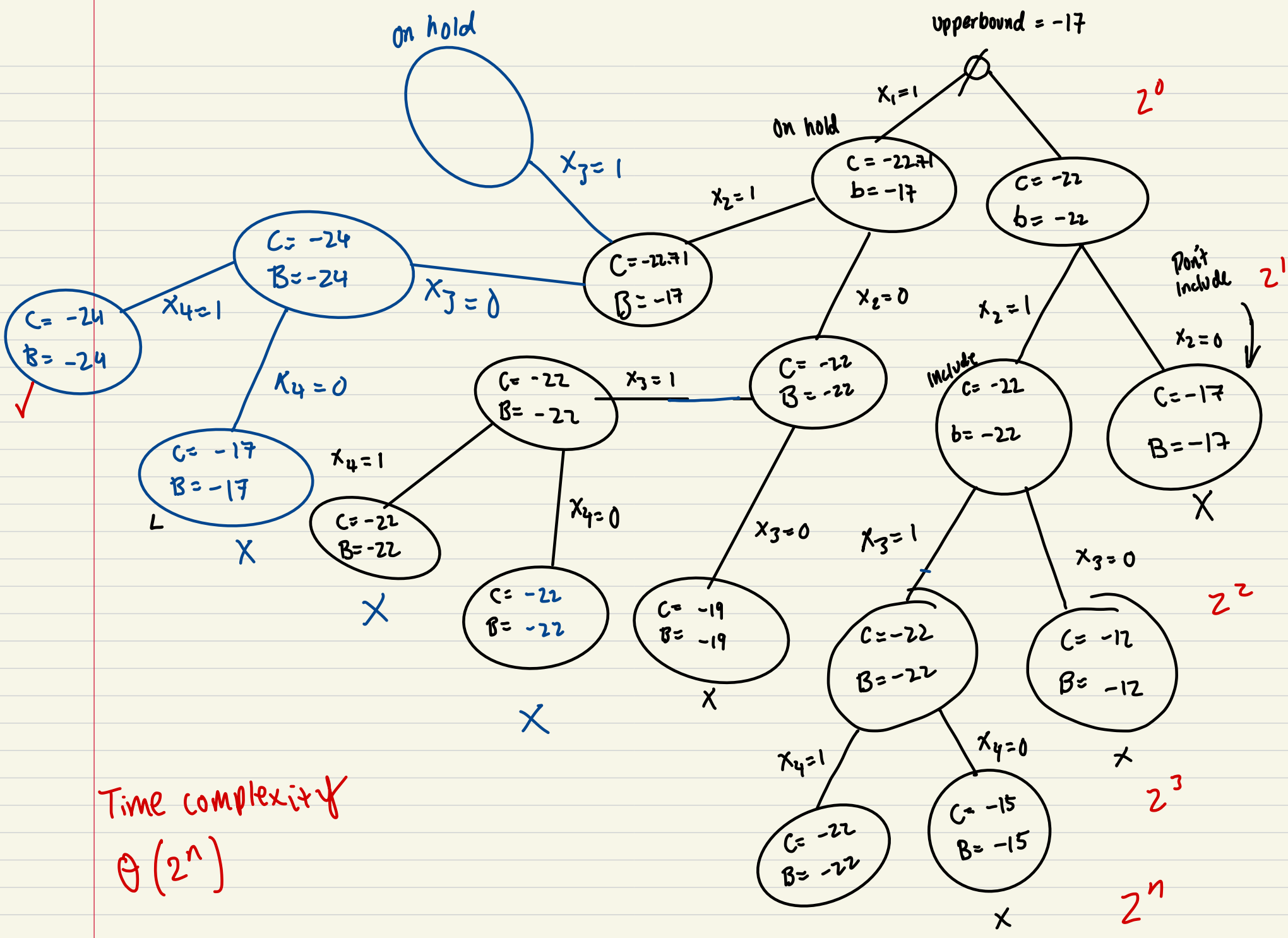
bound = -17



Solution

{ #1, #2, X, #4 } $

$24



Time complexity

$\Theta(2^n)$

Items = } 1, 2, 3, 4 }

Weights = } 5, 3, 7, 2 }

Profit = } 12, 5, 10, 7 }

$$\frac{\text{Profit}}{\text{Weight}} = \left\{ \frac{12}{5}, \frac{5}{3}, \frac{10}{7}, \frac{7}{2} \right\}$$

2.4, 1.66, 1.42, 3.5

x

$\frac{10}{7}$ (2kg) =

1.42(2kg) = 2.86

Sol = } #4, #1, #2

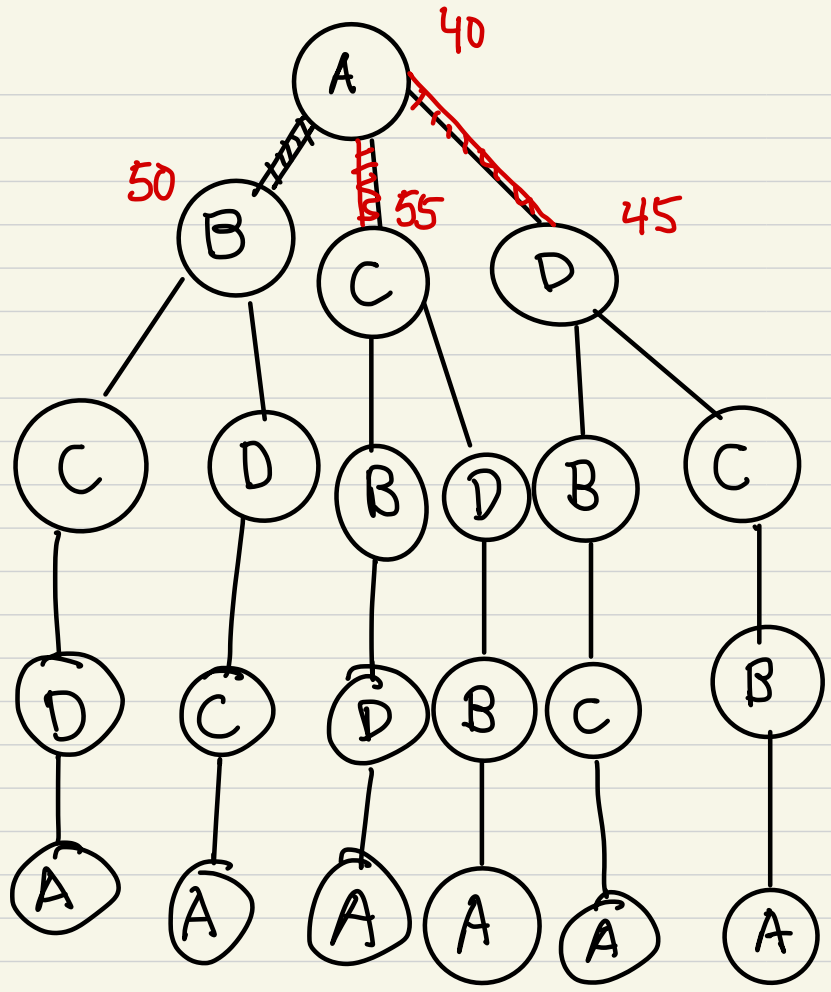
$24 + $ 2.86

= $26.86

	A	B	C	D
A	∞	10	0	5
B	10	∞	0	35
C	0	0	∞	0
D	5	35	0	∞

Bound (A,B)

	A	B	C	D	minRow
A	∞	∞	∞	∞	∞
B	∞	∞	0	35	0
C	0	∞	∞	0	0
D	5	∞	0	∞	0
minCol	0	∞	0	0	



Bound(A) = $\sum$ minRow + minCol

30 + 10 = 40

Bound (A,C)

Bound(A,B) = cost(A,B) + cost(A) + Reduction

10 + 40 + 0

= 50

	A	B	C	D	minRow
A	∞	∞	∞	∞	∞
B	0	∞	∞	25	10
C	∞	0	∞	0	0
D	0	30	∞	∞	5
Min Col	0	0	∞	0	

Bound(A,C) = cost(A,C) + cost(A) + Reduction

0 + 40 + 15

= 55

Bound (A,D)

Bound(A,D) = cost(A,D) + cost(A) + Reduction

5 + 40 + 0

= 45

	A	B	C	D	
A	∞	∞	∞	∞	∞
B	10	∞	0	∞	0
C	0	0	∞	∞	0
D	∞	35	0	∞	0
	0	0	0	∞	

Bound (A, D)

	A	B	C	D	
A	∞	∞	∞	∞	∞
B	10	∞	0	∞	0
C	0	0	∞	∞	0
D	∞	35	0	∞	0
	0	0	0	∞	

→

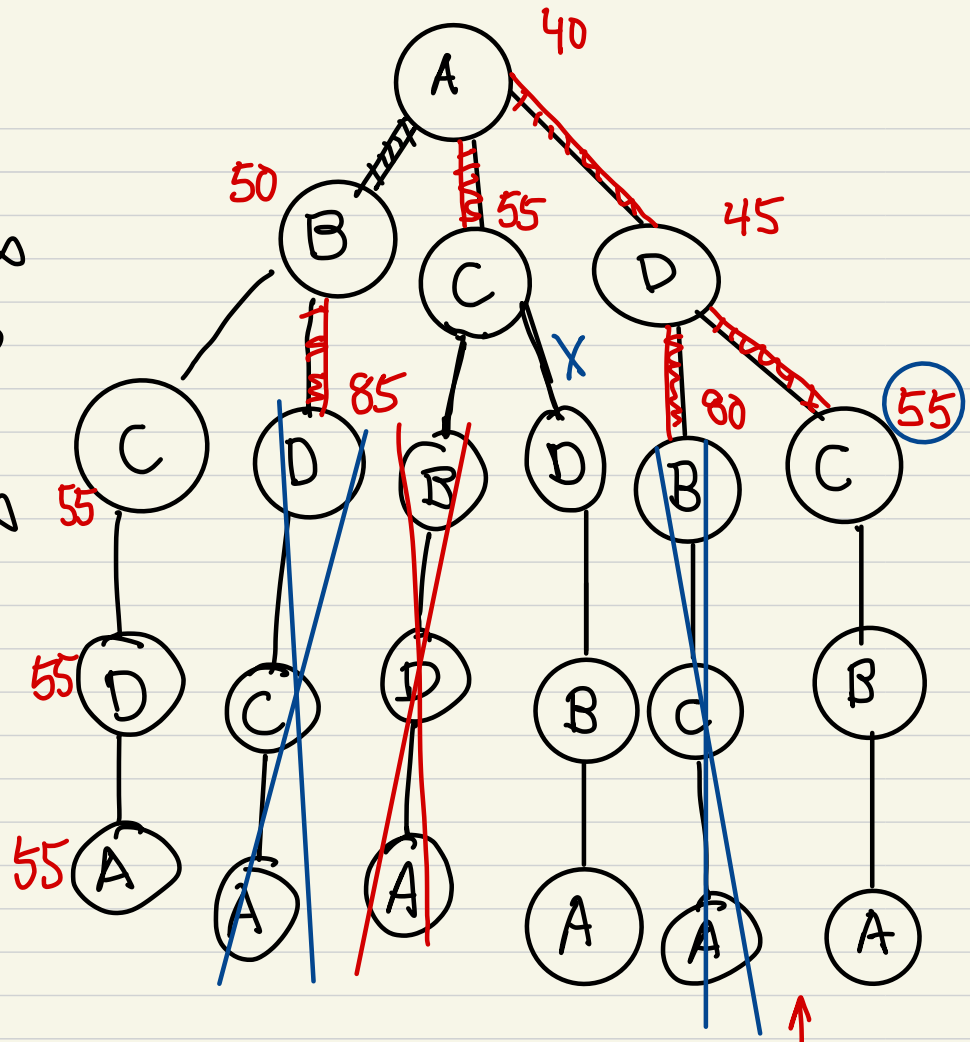
Bound (A, D, B)

	A	B	C	D	
A	∞	∞	∞	∞	∞
B	∞	∞	0	∞	0
C	0	∞	∞	∞	0
D	∞	∞	∞	∞	∞
	0	∞	0	∞	

Bound (A, D, B) = cost (D, B) + cost (D) + Reduction

35 + 45 + 0

= 80



80 > 55

So 80 is out

Bound (A, D, C)

	A	B	C	D	
A	∞	∞	∞	∞	∞
B	0	∞	∞	∞	10
C	∞	0	∞	∞	0
D	∞	∞	∞	∞	∞
	0	0	∞	∞	

=10

Why do you set 10 to 0 at Row B once you put 10 in MinRow

read from row take Min value, then make Reduction, then take Min (0/

Bound (A, D, C) = cost (D, C) + cost (D) + Reduction

0 + 45 + 10

= 55

---

now A → B → D

Bound (A, B)

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	0	35
C	0	∞	∞	0
D	5	∞	0	∞

→

Bound (A, B, D)

	A	B	C	D	minRow
A	∞	∞	∞	∞	
B	∞	∞	∞	∞	
C	0	∞	∞	∞	
D	∞	∞	0	∞	
minCol					

Bound (A, B, D) = cost (B, D) + cost (B) + Reduction

35 + 50 + 0

A − D − C − B − A = 55

Hw #3 notes

1. All the Index i and j for that Sum of P_k are Unique

$M[i=0][j=2] + M[i=2][j=4] + M[i=4][j=3] + M[i=3][j=0]$

3.

$M[i=0][j=2] + M[i=2][j=4] + M[i=4][j=3] + M[i=3][j=0]$

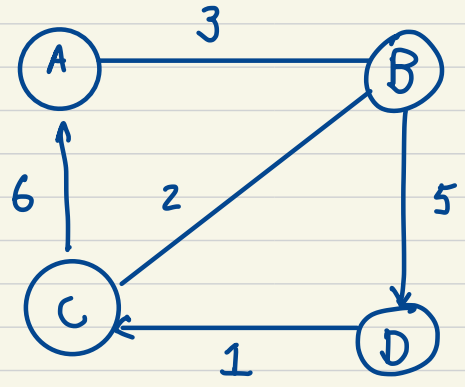
4.

$M[i=0][j=2] + M[i=2][j=4] + M[i=4][j=3] + M[i=3][j=0]$

$i = 0$, $j = 0$

Sample problem

	A	B	C	D
A	0	3	6	0
B	3	0	2	5
C	6	2	0	1
D	0	5	1	0



$M[0][1] + M[1][2] + M[2][3] + M[3][4] + M[4][5] + M[5][0]$

$7 + 5 + 4 + 2 + 3 + 8$

$= 29$

$M[0][5] + M[5][1] + M[1][4] + M[4][3] + M[3][2] + M[2][0]$

$8 + 6 + 9 + 2 + 4 + 1$

$= 30$

CSC510-04 Analysis of Algorithms
Branch and Bound Algorithms

Jose Ortiz
jortizco@sfsu.edu

Overview

- 0-1 Knapsack
  - Best First. Also know as Least Cost (LC)
  - Depth First
  - Breadth First
- Traveling Salesman

What are branch and bound algorithms ?

- Branch and bound algorithms exploits the Dynamic Programing Principle. However, it makes a passive use of this principle. Branch and bound relies on the bounding principle from optimization.

- Conceptual example:

  - Let's say that you are exploring two sets $S1$ and $S2$ and the rules for the upper and lower bounds of the optimal solutions are known. Then, if $S1$ includes the lower and upper bounds, and if the upper bound of the solutions from $S1$ is less than the lower bound for the solutions of $S2$, then it is not worth exploring if $S2$ contains optimal solutions.

0-1 Knapsack Problem
Branch and Bound

0-1 Knapsack Problem

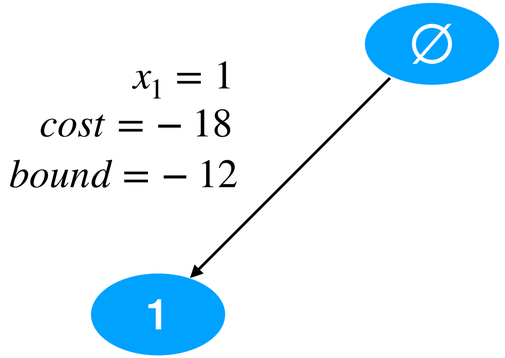
Given a list of items $item = \{1,2,3,4\}$, their weights $w = \{3,4,2,5\}$ and their profit $p = \{3,8,1,15\}$. Determine the number of items to include in a knapsack so that the total weight is at most $w = 11$, and the total profit is as large as possible. (maximized profit)

0-1 Knapsack Problem Best-Case

$W = 11$

Items	1	2	3	4
Profit	3	8	1	15
Weigths	3	4	2	5

$upper\ bound = -12$



Rules :

Branch chosen : *best case*

Bounds :

$$cost = (-1)\sum_{i}^{n} P_i + (fractional)$$

$$bound = (-1)\sum_{i}^{n} P_i$$

X_1 *included* :

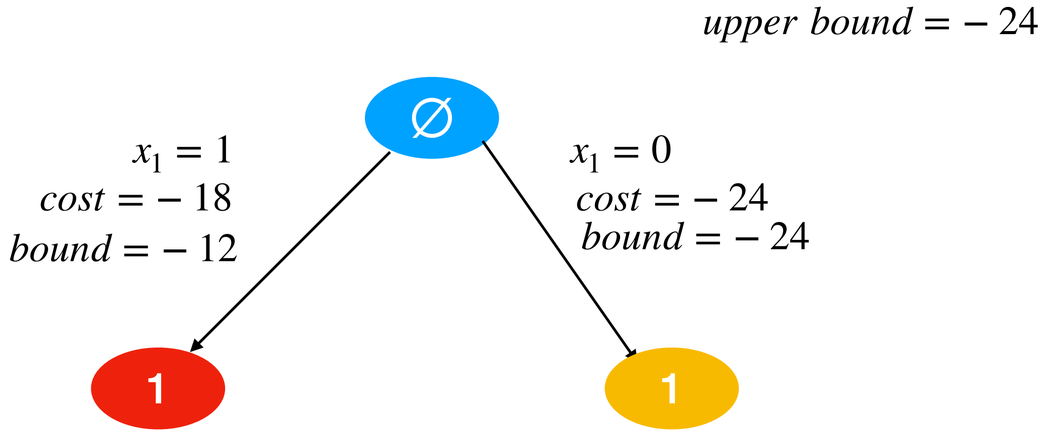
$$cost = (-1)(3 + 8 + 1 + (\frac{15}{5}(2)) = -18$$

$$bound = (-1)(3 + 8 + 1) = -12$$

0-1 Knapsack Problem Best-Case (minimization)

$W = 11$

Items	1	2	3	4
Profit	3	8	1	15
Weigths	3	4	2	5



Rules :

Branch chosen : *best case*

Bounds :

$$cost = (-1)\sum_{i}^{n} P_i \ + (fractional)$$

$$bound = (-1)\sum_{i}^{n} P_i$$

For next iteration, we consider only the minimum upper bound

X_1 *not included* :

$$cost = (-1)(0 + 8 + 1 + 15) = -24$$

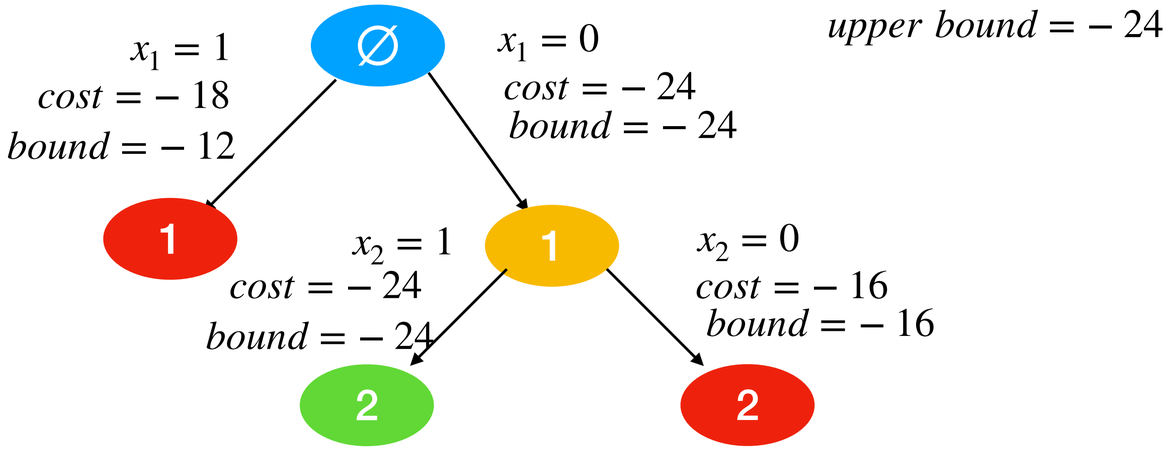
$$bound = (-1)(0 + 8 + 1 + 15) = -24$$

$$upper\_bound = min\{bound(x_1 = 1), bound(x_1 = 0)\} = -24$$

0-1 Knapsack Problem Best-Case (minimization)

$$W = 11$$

Items	1	2	3	4
Profit	3	8	1	15
Weigths	3	4	2	5



Rules :

Branch chosen : *best case*

Bounds :

$$cost = (-1)\sum_{i}^{n} P_i \ + (fractional)$$

$$bound = (-1)\sum_{i}^{n} P_i$$

X_2 *not included* :

$$cost = (-1)(0+0+1+15) = -16$$

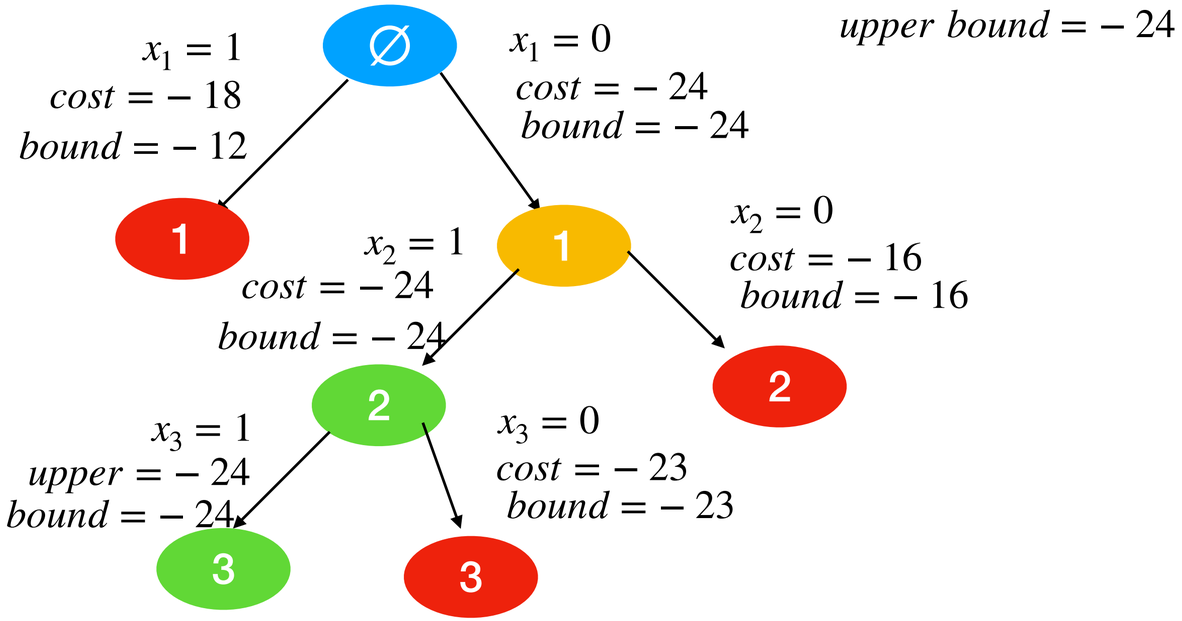
$$bound = (-1)(0+0+1+15) = -16$$

$$upper\_bound = min\{bound(x_1 = 1), bound(x_1 = 0)\} = -24$$

0-1 Knapsack Problem Best-Case (minimization)

$$W = 11$$

Items	1	2	3	4
Profit	3	8	1	15
Weigths	3	4	2	5



Rules :

Branch chosen : *best case*

Bounds :

$$cost = (-1)\sum_{i}^{n} P_i \ + (fractional)$$

$$bound = (-1)\sum_{i}^{n} P_i$$

X_3 *not included* :

$$cost = (-1)(0 + 8 + 0 + 15) = -23$$

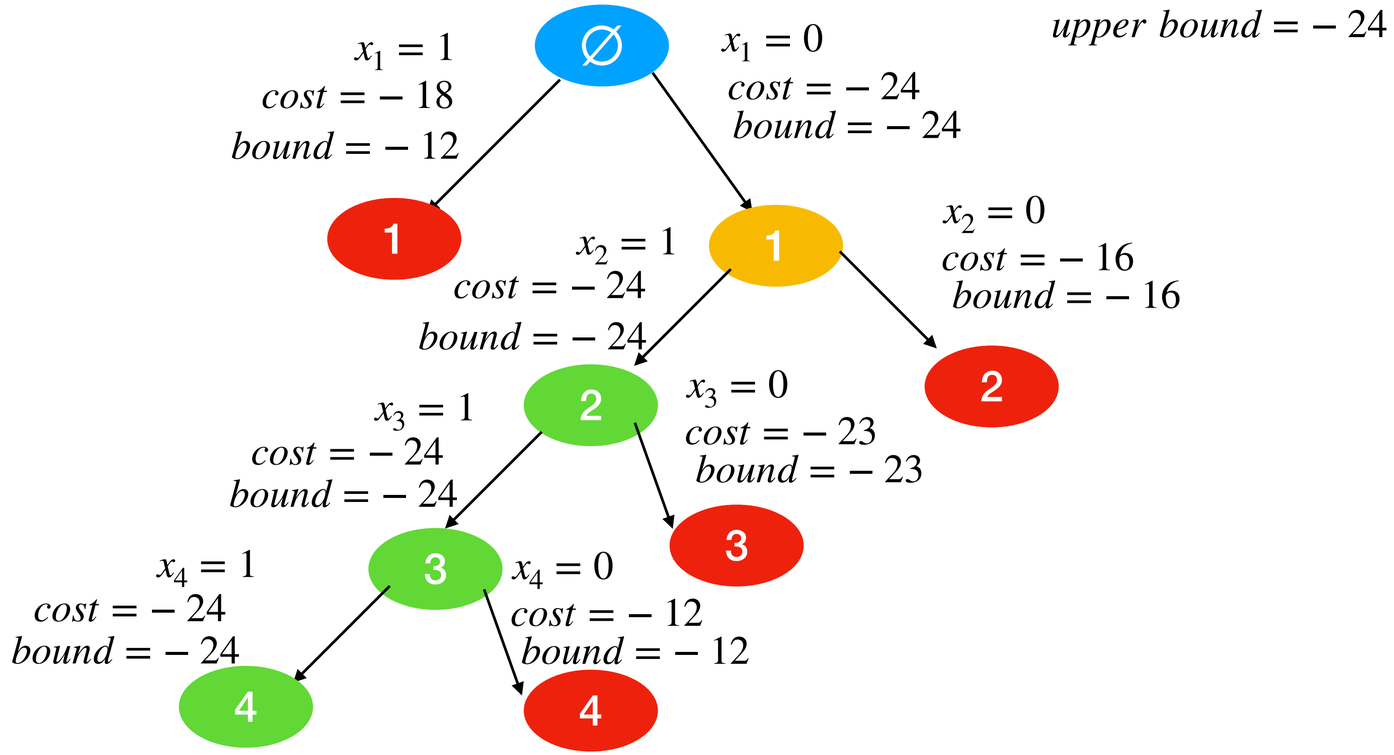
$$bound = (-1)(0 + 8 + 0 + 15) = -23$$

$$upper\_bound = min\{bound(x_1 = 1), bound(x_1 = 0)\} = -24$$

0-1 Knapsack Problem Best-Case (minimization)

$$W = 11$$

Items	1	2	3	4
Profit	3	8	1	15
Weigths	3	4	2	5



Rules :

Branch chosen : *best case*

Bounds :

$$cost = (-1)\sum_{i}^{n} P_i \ + (fractional)$$

$$bound = (-1)\sum_{i}^{n} P_i$$

X_4 *not included* :

$$cost = (-1)(3 + 8 + 1 + 0) = -12$$

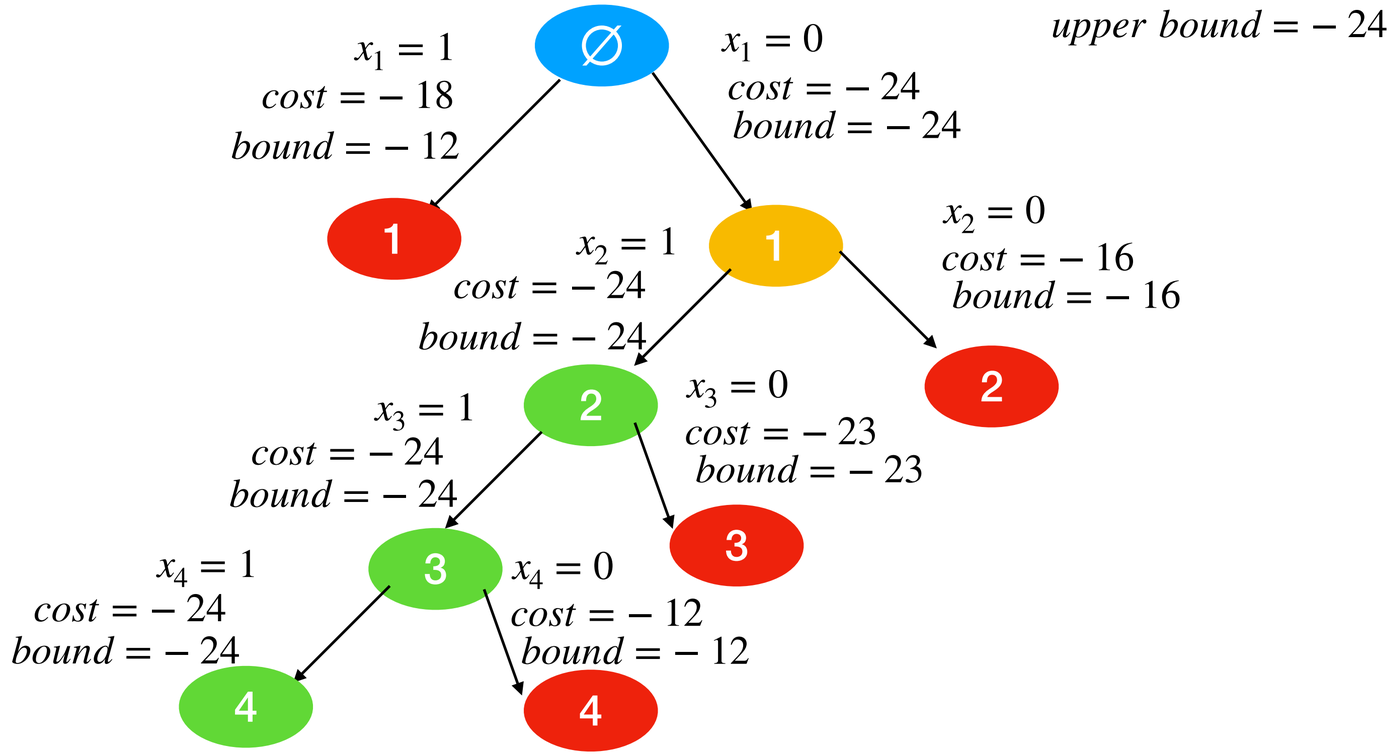
$$upper = (-1)(3 + 8 + +1 + 0) = -12$$

$$upper\_bound = min\{bound(x_1 = 1), bound(x_1 = 0)\} = -24$$

0-1 Knapsack Problem Best-Case (minimization)

$$W = 11$$

Items	1	2	3	4
Profit	3	8	1	15
Weigths	3	4	2	5



Rules :

Branch chosen : *best case*

Bounds :

$$cost = (-1)\sum_{i}^{n} P_i \ + (fractional)$$

$$bound = (-1)\sum_{i}^{n} P_i$$

$$Items\ included = \{2,3,4\} \quad Total\ weight = \sum_{i}^{n} W_{2,3,4} = 11$$

$$Total\ profit = \sum_{i}^{n} P_{2,3,4} = 24$$

0-1 Knapsack Problem Best-Case (minimization) Time complexity Analysis

- This algorithm using the best-case technique has its worst time complexity $O(2^n)$ when we need to fully calculate the entire tree. However, at best, the time complexity to find a solution becomes linear since it is inspecting the best-cases for every path, and rejecting the rest of them.

- Still in the worst case, this algorithm performs better than backtracking because some nodes are not explored based on their bounds and infeasibility

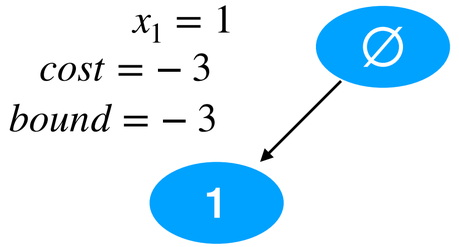
- This algorithm is better implemented using a priority queue to keep track of the best nodes.

0-1 Knapsack Problem: Depth First

$W = 11$

Items	1	2	3	4
Profit	3	8	1	15
Weigths	3	4	2	5

$upper\ bound = -3$



Rules :

Branch chosen : *best case*

Bounds :

$$cost = (-1)\sum_{i}^{n} P_i + (fractional)$$

$$bound = (-1)\sum_{i}^{n} P_i$$

X_1 *included* :

$cost = (-1)(3) = -3$

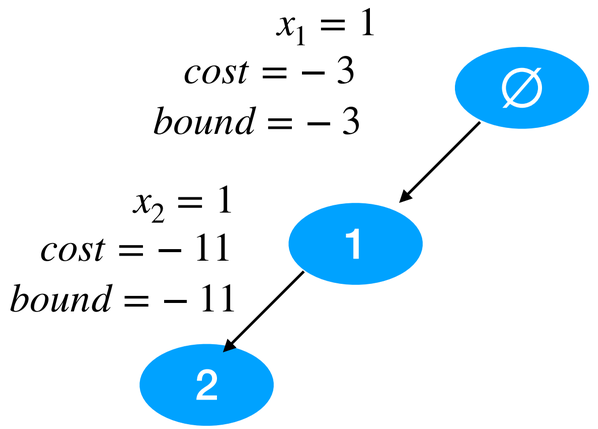
$bound = (-1)(3) = -3$

0-1 Knapsack Problem: Depth First

$W = 11$

Items	1	2	3	4
Profit	3	8	1	15
Weigths	3	4	2	5

$upper\ bound = -11$



Rules :

Branch chosen : *best case*

Bounds :

$$cost = (-1)\sum_{i}^{n} P_i \ + (fractional)$$

$$bound = (-1)\sum_{i}^{n} P_i$$

X_2 *included* :

$cost = (-1)(3+8) = -11$

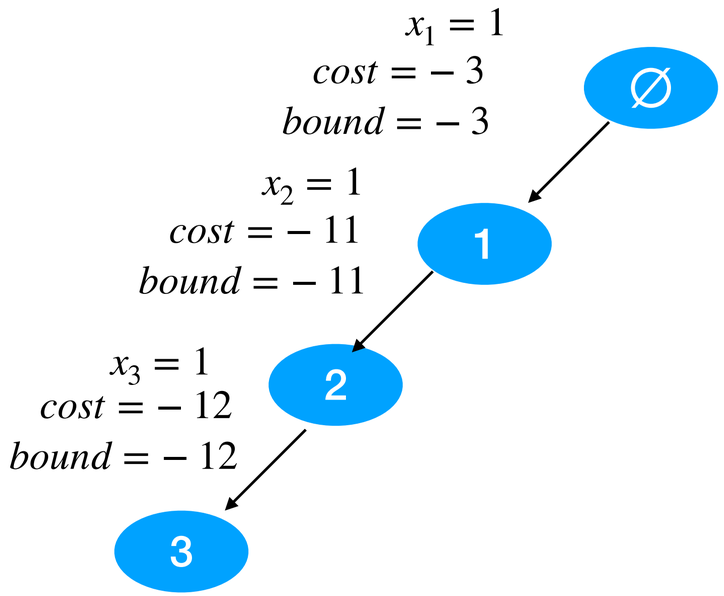
$bound = (-1)(3+8) = -11$

0-1 Knapsack Problem: Depth First

$W = 11$

Items	1	2	3	4
Profit	3	8	1	15
Weigths	3	4	2	5

$upper\ bound = -12$



Rules :

Branch chosen : *best case*

Bounds :

$$cost = (-1)\sum_{i}^{n} P_i + (fractional)$$

$$bound = (-1)\sum_{i}^{n} P_i$$

X_3 *included* :

$cost = (-1)(3 + 8 + 1) = -12$

$bound = (-1)(3 + 8 + 1) = -12$

0-1 Knapsack Problem: Depth First

$W = 11$

Items	1	2	3	4
Profit	3	8	1	15
Weigths	3	4	2	5

Rules :

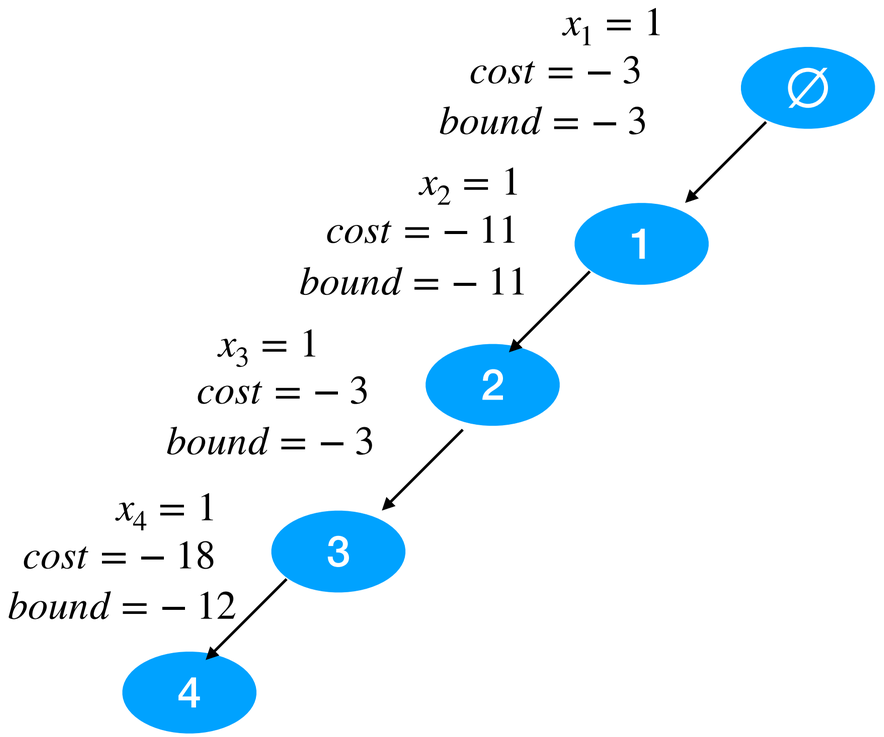
Branch chosen : *best case*

Bounds :

$$cost = (-1)\sum_{i}^{n} P_i \ + (fractional)$$

$$bound = (-1)\sum_{i}^{n} P_i$$

$upper\ bound = -12$



X_4 *included* :

$$cost = (-1)(3 + 8 + 1 + (\frac{15}{5}(2)) = -18$$

$$bound = (-1)(3 + 8 + 1) = -12$$

0-1 Knapsack Problem: Depth First

$W = 11$

Items	1	2	3	4
Profit	3	8	1	15
Weigths	3	4	2	5

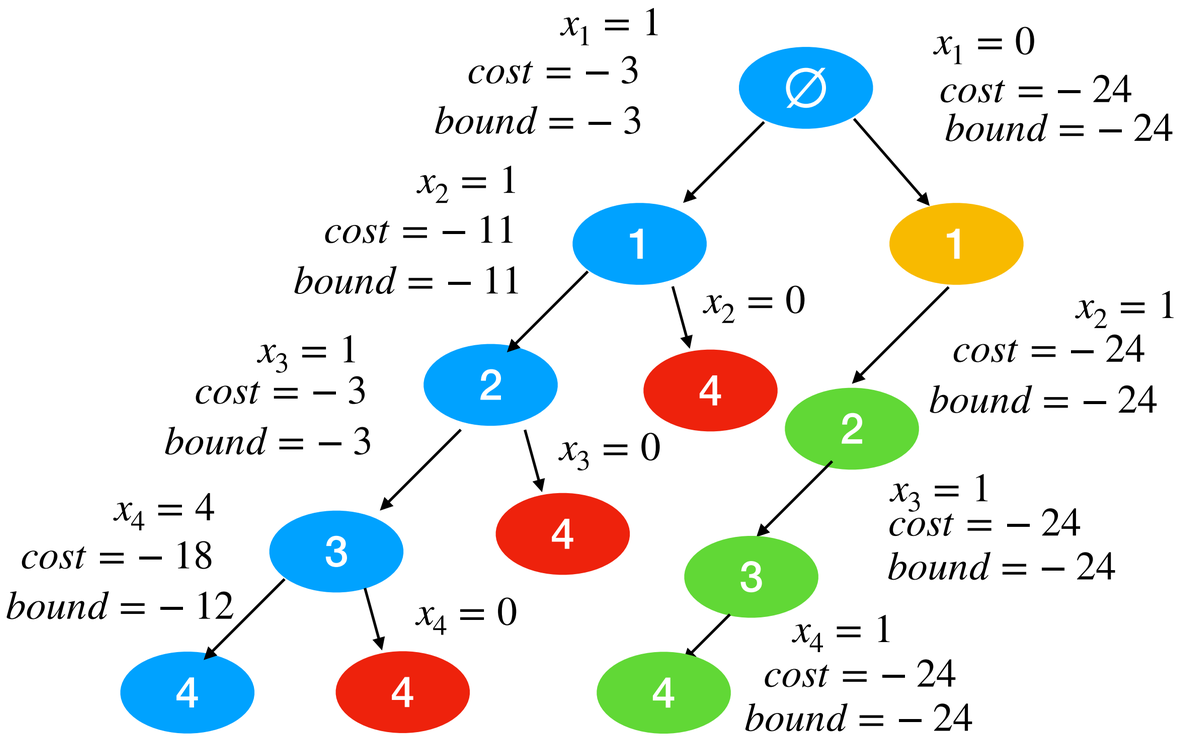
Rules :

Branch chosen : *best case*

Bounds :

$$cost = (-1)\sum_{i}^{n} P_i \ + (fractional)$$

$$bound = (-1)\sum_{i}^{n} P_i$$



X_1 *(not included) and* X_2, X_3, X_4 *(included)* :

$$cost = (-1)(0 + 8 + 1 + 15) = -24$$

$$bound = (-1)(0 + 8 + 1 + 15) = -24$$

$$upper\_bound = min\{bound(x_1 = 1), bound(x_1 = 0)\} = -24$$

0-1 Knapsack Problem: Depth First

$$W = 11$$

Items	1	2	3	4
Profit	3	8	1	15
Weigths	3	4	2	5

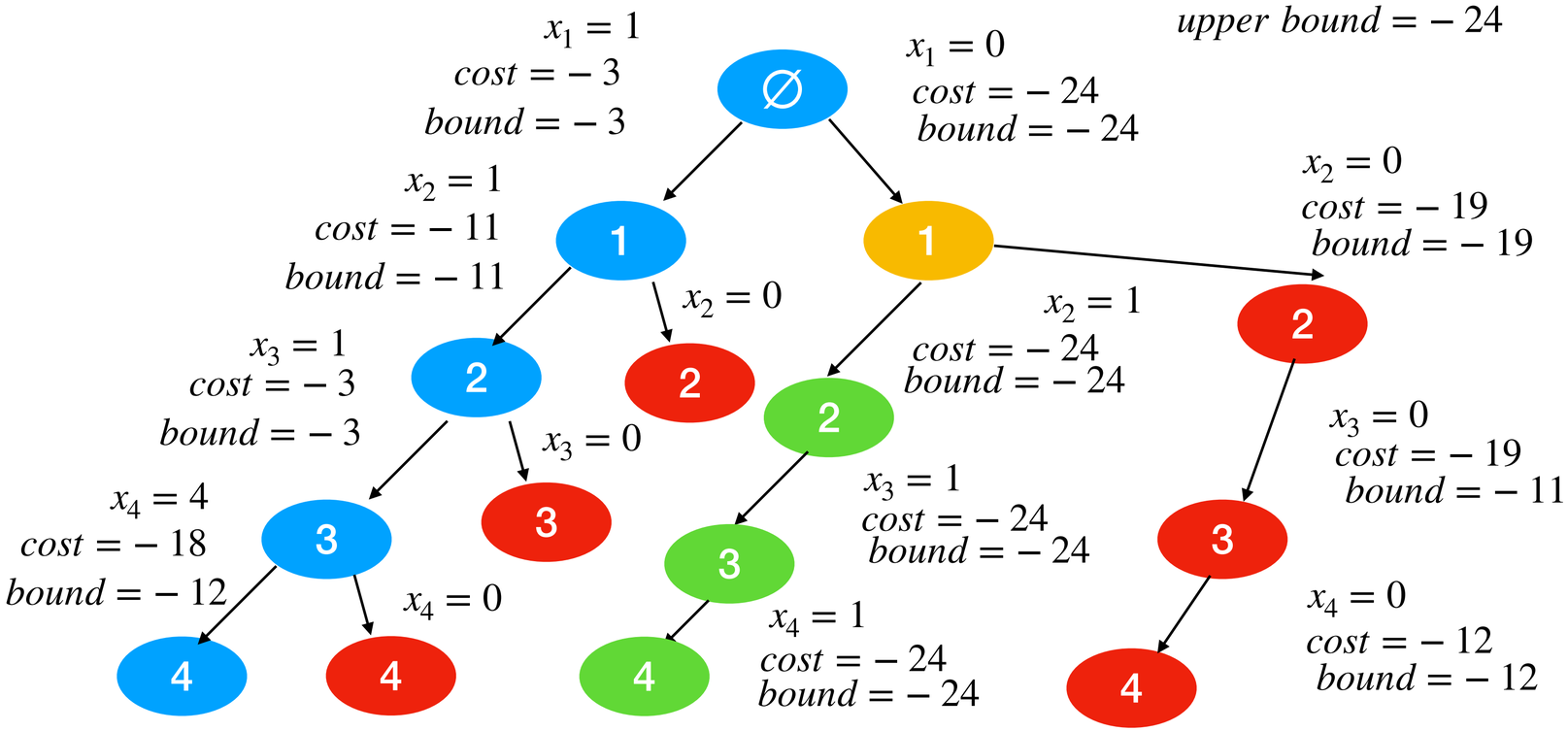
Rules :

Branch chosen : *best case*

Bounds :

$$cost = (-1)\sum_{i}^{n} P_i \; + (fractional)$$

$$bounc = (-1)\sum_{i}^{n} P_i$$



Items included $= \{2,3,4\}$ *Total weight* $= \sum_{i}^{n} W_{2,3,4} = 11$

Total profit $= \sum_{i}^{n} P_{2,3,4} = 24$

0-1 Knapsack Problem Depth-First Time complexity Analysis

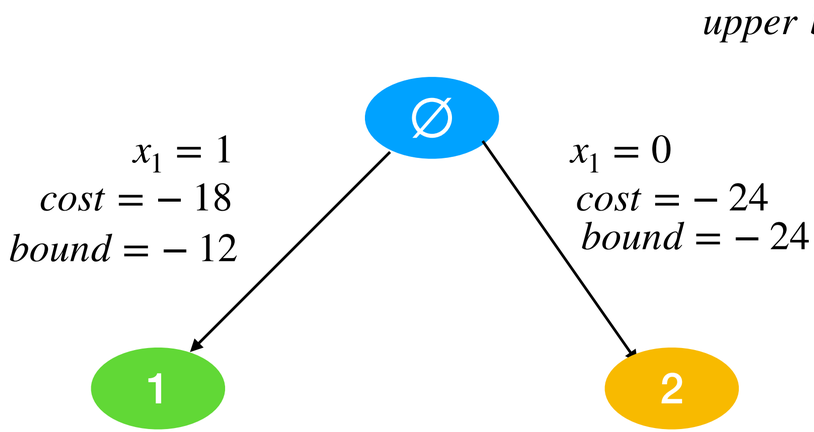
- Since this approach needs to first compute all the included nodes (same as backtracking) the bound is only computed on the last included node, and the number of computations increases in relation to the branch and bound (best-case)

- Worst case scenario for this approach is still $O(2^n)$

0-1 Knapsack Problem: Breadth-First

$W = 11$

Items	1	2	3	4
Profit	3	8	1	15
Weigths	3	4	2	5



Rules :

Branch chosen : *best case*

Bounds :

$$cost = (-1)\sum_{i}^{n} P_i \ + (fractional)$$

$$bound = (-1)\sum_{i}^{n} P_i$$

X_1 *not included* :

$$cost = (-1)(0 + 8 + 1 + 15) = -24$$

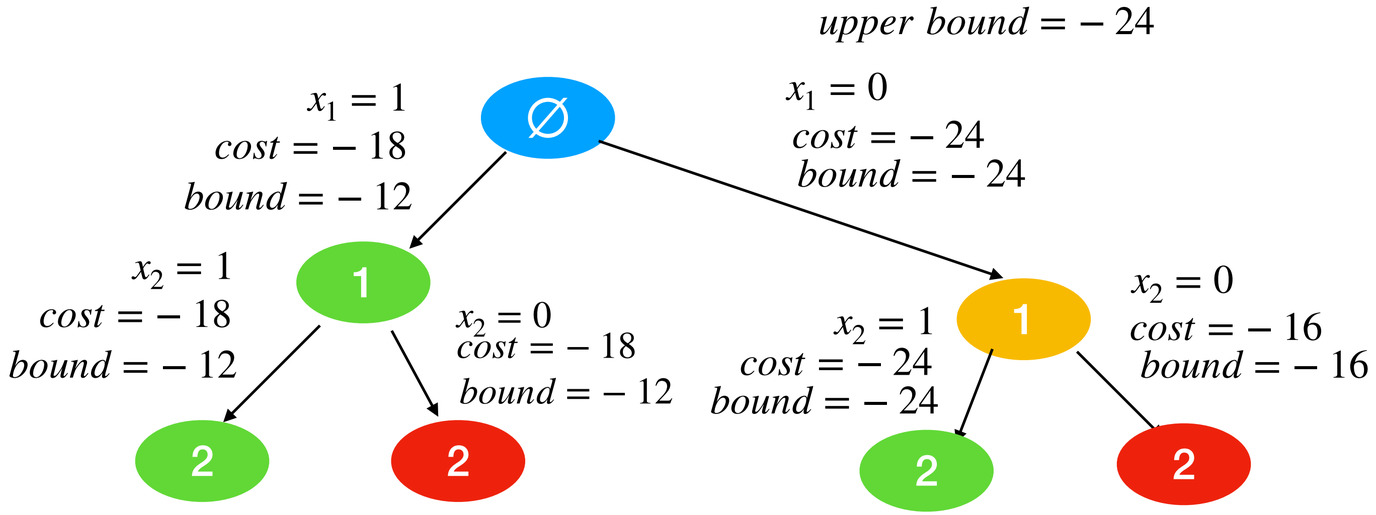
$$bound = (-1)(0 + 8 + 1 + 15) = -24$$

$$upper\_bound = min\{bound(x_1 = 1), bound(x_1 = 0)\} = -24$$

0-1 Knapsack Problem: Breadth-First

$$W = 11$$

Items	1	2	3	4
Profit	3	8	1	15
Weigths	3	4	2	5



Rules :

Branch chosen : *best case*

Bounds :

$$cost = (-1)\sum_{i}^{n} P_i \ + (fractional)$$

$$bound = (-1)\sum_{i}^{n} P_i$$

0-1 Knapsack Problem: Breadth-First

$W = 11$

Items	1	2	3	4
Profit	3	8	1	15
Weigths	3	4	2	5

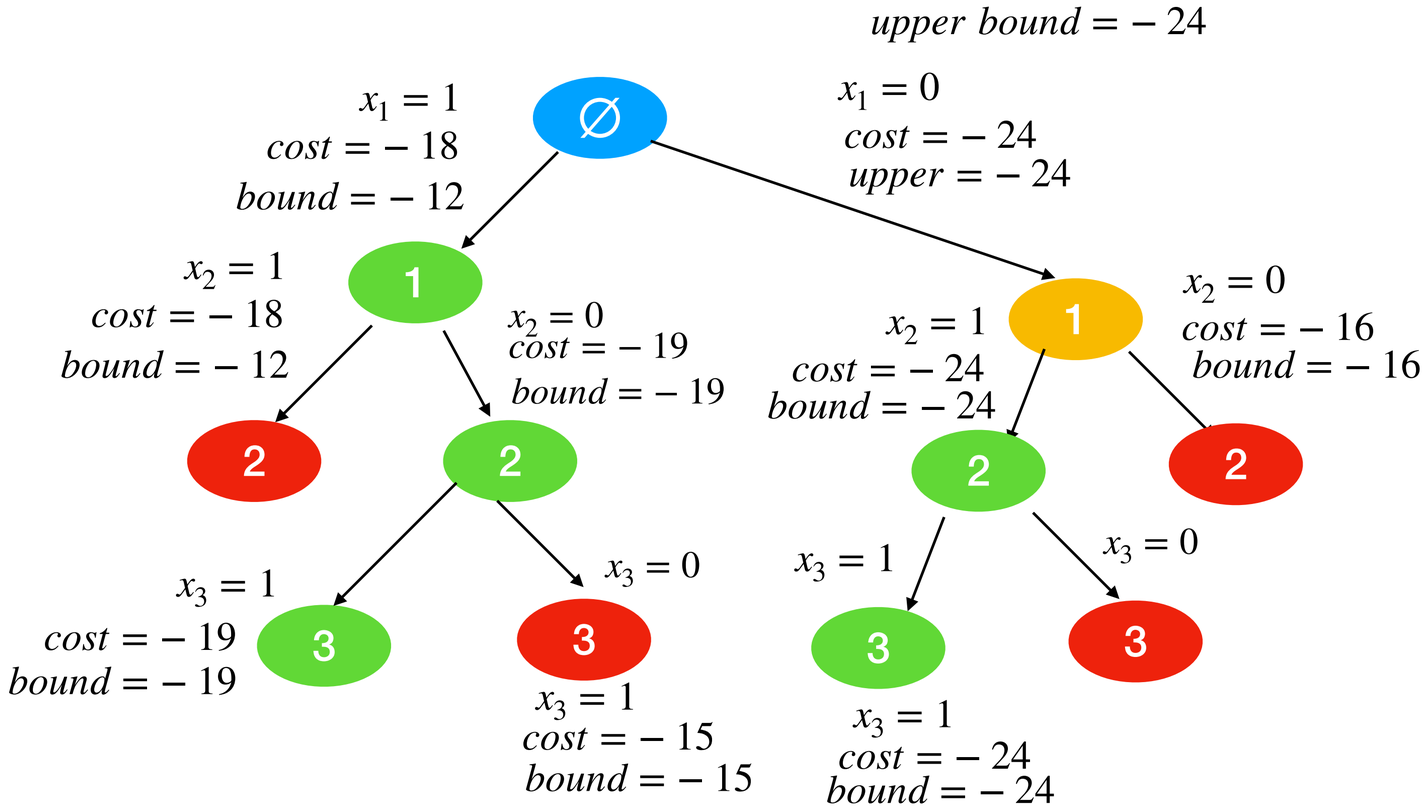
Rules :

Branch chosen : *best case*

Bounds :

$$cost = (-1)\sum_{i}^{n} P_i \; + (fractional)$$

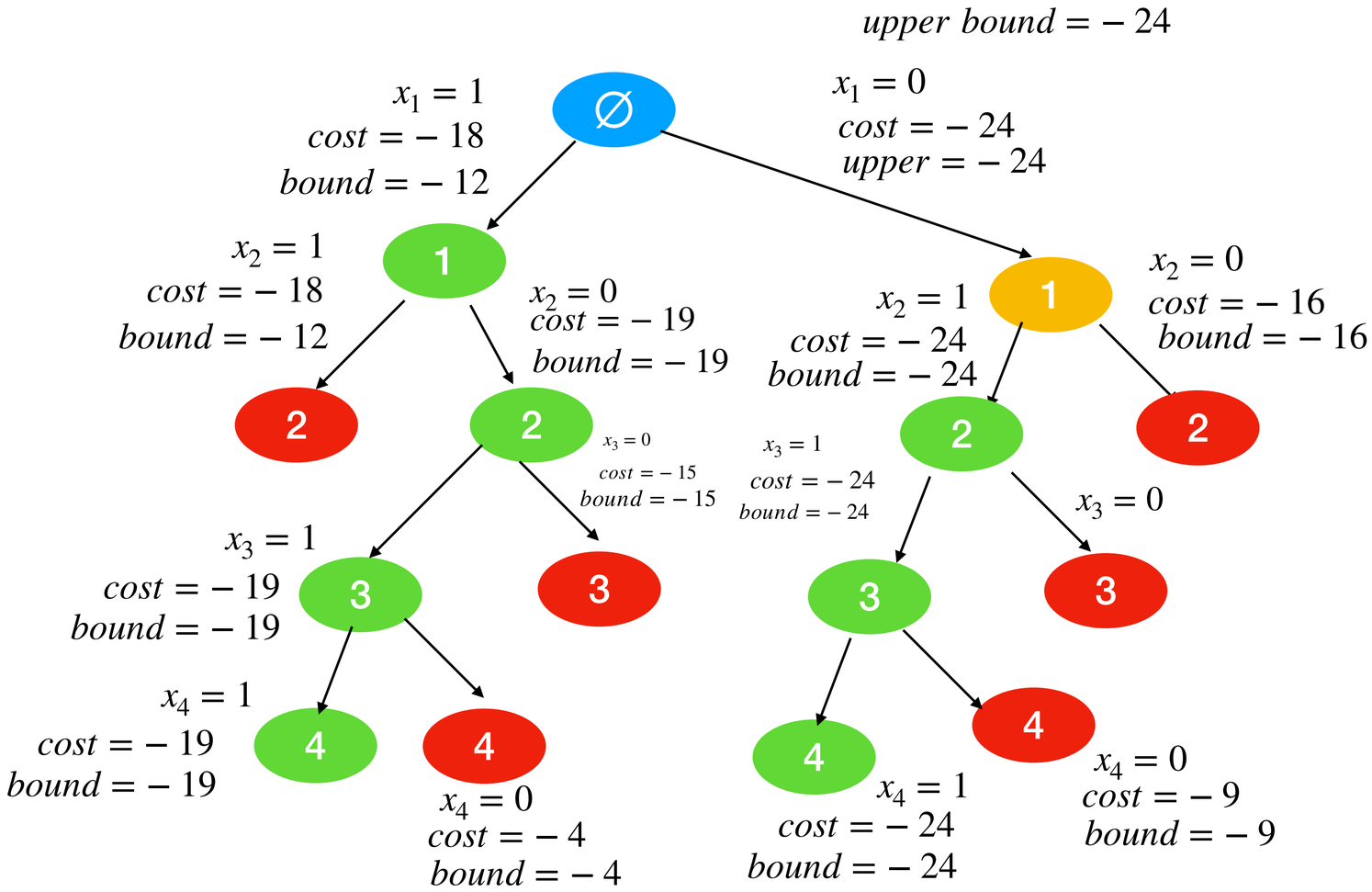
$$bound = (-1)\sum_{i}^{n} P_i$$



0-1 Knapsack Problem: Breadth-First

$$W = 11$$

Items	1	2	3	4
Profit	3	8	1	15
Weigths	3	4	2	5



Rules :

Branch chosen : *best case*

Bounds :

$$cost = (-1)\sum_{i}^{n} P_i \ + (fractional)$$

$$upper = (-1)\sum_{i}^{n} P_i$$

$$Items\ included = \{2,3,4\} \quad Total\ weight = \sum_{i}^{n} W_{2,3,4} = 11$$

$$Total\ profit = \sum_{i}^{n} P_{2,3,4} = 24$$

0-1 Knapsack Problem Breadth-First Time complexity Analysis

- This approach needs to compute nodes at each level. This means that until the algorithm doesn't reach the last level, a decision cannot be made. Worst case scenario for this approach is still $O(2^n)$

0-1 Knapsack Problem Brach and Bound Summary

From all the brach and bound approaches to solve this problem, the best-case approach is the best suited since it only considers good nodes with maximum profit and rejects all the unfeasible paths. Thus, by using the best-case approach, the number of computations made by the algorithm decreases in relation to Depth-First and Breadth-First ones.

Traveling Salesperson Problem
Branch and Bound
Least Cost

TSP Branch and Bound

- Problem Statement
  - Given a a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the original city?
  - Brute force (next slide) will compute all the possible permutations in order to get the shortest possible route in $O(n!)$. Not good!
  - We can do slightly better using dynamic programming since it will take only into account the minimum shortest paths computed so far in $O(n^2 2^n)$. This may not seem a great improvement. However, it will make the problem more feasible with graphs of at most 20 nodes approximately using a typical computer.
  - The branch and bound technique for solving this problem uses a breadth-first approach to display the nodes, and best-case to select the minimum cost path for each node

TSP Branch and Bound

Here is the adjacently matrix for the traveling salesman problem representing a graph of four vertices

Find the shortest possible route that visits each vertex exactly once, and returns to the source vertex. Consider for this problem source vertex to be V_0 using Branch and Bound (Least Cost)

	A	B	C	D
A	∞	25	10	15
B	25	∞	10	45
C	10	10	∞	5
D	15	45	5	∞

TSP Branch and Bound (Least Cost)

1. For each row, compute the minimum value

	A	B	C	D	Min
A	∞	25	10	15	10
B	25	∞	10	45	10
C	10	10	∞	5	5
D	15	45	5	∞	5

2. For each cell, reduce its value using the following formula $ReducedVal = ActualVal - MinVal$

	A	B	C	D
A	∞	15	0	5
B	15	∞	0	35
C	5	5	∞	0
D	10	40	0	∞

3. For each column, compute the minimum value

	A	B	C	D
A	∞	15	0	5
B	15	∞	0	35
C	5	5	∞	0
D	10	40	0	∞
Min	5	5	0	0

4. For each column, reduce its value using the following formula $ReducedVal = ActualVal - MinVal$

	A	B	C	D
A	∞	10	0	5
B	10	∞	0	35
C	0	0	∞	0
D	5	35	0	∞

$$ReducedCost(A) = \sum MinCol + \sum MinRow = 10 + 30 = 40$$

TSP Branch and Bound (Least Cost)

Reduced Matrix

	A	B	C	D
A	∞	10	0	5
B	10	∞	0	35
C	0	0	∞	0
D	5	35	0	∞

A -> D

	A	B	C	D
A	∞	∞	∞	∞
B	10	∞	0	∞
C	0	0	∞	∞
D	∞	35	0	∞

A -> B

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	0	35
C	0	∞	∞	0
D	5	∞	0	∞

A -> D ->B

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	0	∞
C	0	∞	∞	∞
D	∞	∞	∞	∞

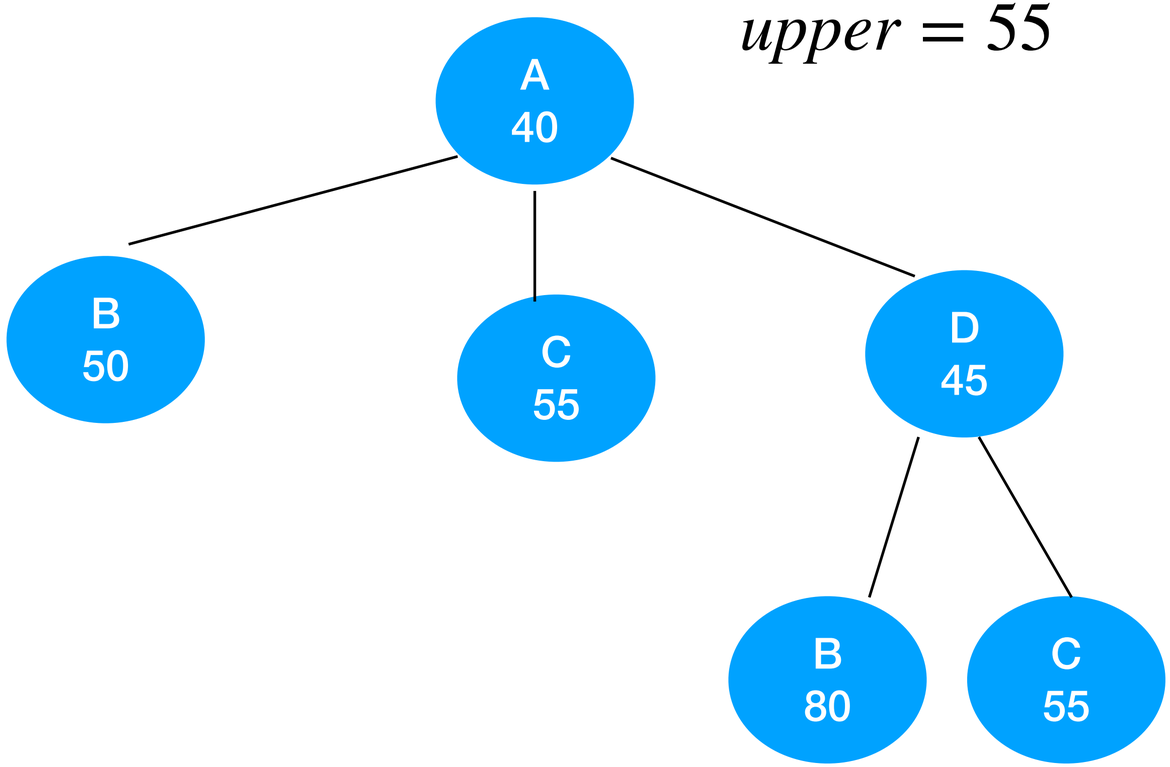
A ->C

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	25
C	∞	0	∞	0
D	0	30	∞	∞

A -> D -> C

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	0	∞
C	∞	∞	∞	∞
D	∞	∞	∞	∞

$upper = 55$



Minimum Path: $A - D - C - B - A = 55$

TSP Branch and Bound Summary

- The best-case approach together with a breadth-first search give pretty good results in this problem because we reject unpromising vertices before exploring in deep the tour for that vertex.

- The time complexity of this problem is still similar to the brute force one. However, in practice, branch and bound performs better depending on the bounding function given

- TSP still is one of the most interesting problems in algorithms and one of the most researched.

Date: 10/23/23

Pointer

pi	wi	i	0	1	2	3	4	5	6	7	8	9
$0	0kg	0	$0	$0	$0	$0	$0	$0	$0	$0	$0	$0
$50	5kg	1	$0	$0	$0	$0	$0	$50	$50	$50	$50	$50
$20	3kg	2	$0	$0	$0	$20	$20	$50	$50	$50	$70	$70
$30	4kg	3	$0	$0	$0	$20	$30	$50	$50	$50	$70	$80
$25	2kg	4	$0	$0	$25	$25	$30	$50	$55	$75	$75	$80
$5	1kg	5	$0	$5	$25	$30	$30	$50	$55	$75	$80	$80

0kg {∅}
5kg {1}
3 {2}
8kg {1,2} = 5+3=8
4kg {3}
9kg {1,3} = 5+4=9
7kg (2,3) = 3+4=7
NO (1,2,3] = 5+3+4=X

$M[4][9] = Max(M[3][9], M[3][9-2] + P[4])$
(i, w)

We don't include the item

We include the item

Max ($80 or $50 + $25)
$75

formula → $M[i][w] = Max(M[i-1][w], M[i-1][w-wi] + P[i])$

Ex: $M[2][8] = Max(M[1][8], M[1][8-3] + P[2])$
$50 , $50 + $20 = $70

Dynamic programing is important because it finds the most optimize solution

→ How to find the items that conform this solution

Sol = { #3 , #1 , ∅ }
9-4 = 5kg
5-5 = 0kg

Empty set always there

Why this ?
because { #3, #1, ∅ } is
better than { #1, #2, #5, ∅ }
$80
$75

Psuedocode

Matrix (n+1) (w+1)

(n+1)(w+1) → Initialize matrix to 0

(n+1) → for each i in matrix:

(w+1) → for each w in matrix

1 → if w < w[i] then M[i][w] = M[i-1][w]

1 → else M[i][w] = Max(M[i-1][w], M[i-1][w-w[i]] + P[i])

$t(n) = (n+1)(w+1) + (w+1) = nw + n + w + 1 + (w+1) \simeq O(nw)$

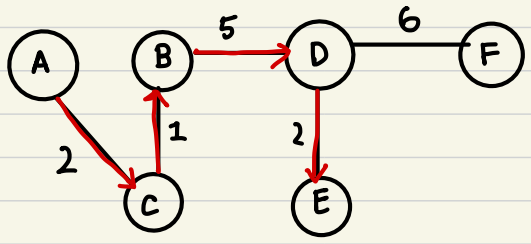
Dijkstra Algorithm (greedy)

10/30/2023

Find the minimum path from the source node to all the other nodes

{V,W}	B	C	D	E	F
A	A,4	A,2	A,∞	A,∞	A,∞
A,C	C,3	X	C,10	C,12	C,∞
A,C,B	X	X	B,8	C,12	B,∞
A,C,B,D	X	X	X	D,10	D,14
A,C,B,D,E	X	X	X	X	D,14

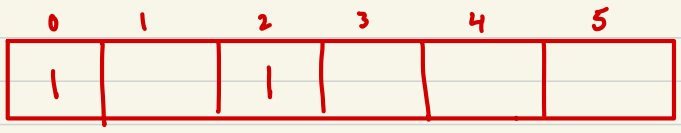
Solution



Destination	path	cost
B	{A,C,B}	3
C	{A,C}	2
D	{A,C,B,D}	8
→ E	{A,C,B,D,E}	10
F	{A,C,B,D,F}	14

$cost(A, V_i) = Min(\{A, V_i\}, \{A, \{V_k\}, V_i\})$

Visited



space complexity

$t(n) = n + n^2$

$= O(n^2)$

time complexity

$O(n^2)$

Min {A,B} , {A,C,B}

4 , 2 + 1 = 3

= 3

Min {A,D} , {A,C,D}

∞ , 2 + 8 = 10

= 10

Min {A,E} , {A,C,E}

∞ , 2 + 10 = 12

= 12

Min {A,F} , {A,C,F}

∞ , 2 +

min {A,C,D} , {A,C,B,D}

10 , 2+1+5 = 8

= 8

{A,C,E} , {A,C,B,E}

12 , 2 + 1 + ∞

= 12

{A,C,F} , {A,C,B,F}

∞ , 2+1+∞

{A,C,B,E} , {A,C,B,D,E}

12 , 2+1+5+2 = 10

= 10

{A,C,B,F} , {A,C,B,D,F}

∞ , 2+1+5+6 = 14

= 14

{A,C,B,D,F} , {A,C,B,D,E,F}

14 , 2+1+5+2+5 = 15

= 14

Visited	B	C	D	E	F
{A}	A,4	A,2	A,∞	A,∞	A,∞

0	1	2	3	4	5
A	B	C	D	E	F
0	(0,4) ↓ (2,3)	(0,2)	(0,∞)	(0,∞)	(0,∞)

Visited =	A 0	B 1	C 2	D 3	E 4	F 5
	1	0	1	0	0	0

$t(n) = n + n = 2n$

for each V in graph : → N

Compute Dijkstra → N^2

$O(n^3)$

Visited	B	C	D	E	F
{A}	A,4	A,2	A,∞	A,∞	A,∞

0	1	2	3	4	5
A	B	C	D	E	F
0	(0,4) ↓ (2,3)	(0,2)	(0,∞)	(0,∞)	(0,∞)

Visited =

A	B	C	D	E	F
0	1	2	3	4	5
1	0	1	0	0	0

$t(n) = n + n = 2n$

for each v in graph : → N

Compute Dijkstra → N^2

$O(n^3)$

Encryption algorithmn

	Data	codes
user 1	01	0000
user 2	10	1111
user 3	11	

bits → volts

$f(b) = v$

two rules

(1) $f(0) = +1v$

(2) $f(1) = -1v$

use XOR operation

1 XOR 1 = 0

0 XOR 0 = 0

1 XOR 0 = 1

0 XOR 1 = 1

Receiver

$f(b) = v$

$$f\left(\frac{\sum_{i=1}^{c} f_i C_i}{\text{Length(code)}}\right) = \text{bit}$$

+1v, +1v, +1v, +1v

−1v, −1v, −1v, −1v

−1v, −1v, −1v, −1v

$= -4v$

	Data	codes
user 1	01	0000
user 2	10	1010
user 3	11	1100

+1v, +1v, +1v, +1v

−1v, −1v, +1v, +1v

(−1v) + (−1v) + 1v + 1v

$= 0$

	Data	codes
user 1	01	0000
user 2	10	1010
user 3	11	1100

V1 = 00001111

XOR 00000000

00001111

V2 11110000

XOR 10101010

01011010

V3 1111 1111

XOR 1100 1100

00110011

V1	0	0	0	0	1	1	1	1
+1v								
−1v								
V2	0	1	0	1	1	0	1	0
+1v								
−1v								
V3	0	0	1	1	0	0	1	1
+1v								
−1v								
$\sum f_i$	+3v	+1v	+1v	−1v	−1v	+1v	−3v	−1v

+3v, +1v, +1v, −1v, −1v, +1v, −3v, −1v

−1v, +1v, −3v, −1v, −1v, +1v, −1v, +1v

$(-3v) + (1v) + (-1v) + (-1v) = \frac{-4v}{4} = -1v$

$(1v) + (1v) + (3v) - 1v = \frac{4v}{4} = 1v$

f(−1v) = 1 bit

f(+1) = 0 bit

Decode $\left(\frac{\sum_{i=1}^{\text{Len(code)}} f_i C_i}{\text{Length(code)}}\right) = \text{bit}$

$f(1\text{ bit}) = -1v$

$f(0\text{ bit}) = +1v$

$f^{-1}(-1v) = 1\text{ bit}$

$f^{-1}(+1v) = 0\text{ bit}$

Psvedo code

next page

n= # uses

Steps for psvedocode

Sender

1. finding orthogonal codes $O(n^3)$

2. XOR operation
   for every $O(n(d+c))$

   d = Length of data
   c = Length of code

3. Tables of frequencies $O(n(d+c))$
   $nd + nc$

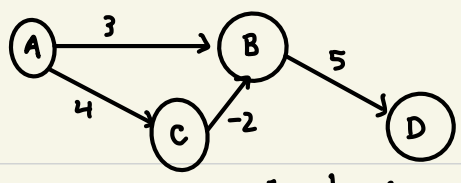
receiver

4. Creating the decoding data process $O(d+c)$

5. Decoding $O(1)$

What approach I have Implemented In this algorithmn

(1) brute force
(2) Divide & conquer
(3) Backtracking (DFS, LC, or BFS)
(4) Dynamic programming
(5) Greedy
(6) none that we covered so far



Bellman - ford (Distance vector version)

P = NP

from

	A	B	C	D
A	0	3	4	∞
B	∞	0	∞	∞
C	∞	∞	0	∞
D	∞	∞	∞	0

to

	A	B	C	D
A	0	∞	∞	∞
B	∞	0	∞	∞
C	∞	-2	0	∞
D	∞	∞	∞	0

	A	B	C	D
A	0	∞	∞	∞
B	∞	0	∞	5
C	∞	∞	0	∞
D	∞	∞	∞	0

	A	B	C	D
A	0	∞	∞	∞
B	∞	0	∞	∞
C	∞	∞	0	∞
D	∞	∞	∞	0

Step 2

	A	B	C	D
A	0	2	4	8
B	∞	0	∞	5
C	∞	-2	0	3
D	∞	∞	∞	0

$Dist_A(B) = \min \begin{cases} A \to B + B \to B, & 3+0 = 3 \\ A \to C + C \to B, & 4+(-2) = 2 \\ A \to D + D \to B, & \infty + \dots \end{cases}$

$= 2$

$Dist_A(C) = \min \begin{cases} A \to C + C \to C = 4+0 \\ A \to B + B \to C = 5+-3 \\ A \to D + D \to C = \infty + x \end{cases}$

$= 4$

$Dist_A(D) = \min \begin{cases} cost(A,D) + Dist_D(D), & \infty + 0 \\ cost(A \to B) + Dist_B(D), & 3+5 = 8 \\ cost(A \to C) + Dist_C(D), & 4+\infty \end{cases}$

$= 8$

$Dist_B(A) = Min \begin{cases} Cost(B,A) + Dist_A(A), & \infty + 0 \\ Cost(B,C) + Dist_C(A), & \infty + \infty \\ Cost(B,D) + Dist_D(A), & 5 + \infty \end{cases}$

$= \infty$

$Dist_B(C) = Min \begin{cases} Cost(B,C) + D, & \infty \\ Cost(B,A) + Dist_A(C), & \infty \\ Cost(B,D) + Dist_D(C), & 5 \end{cases}$

Matrix(n) = Matrix(n)-1

	A	B	C	D
A	0			
B		0		
C			0	
D				0

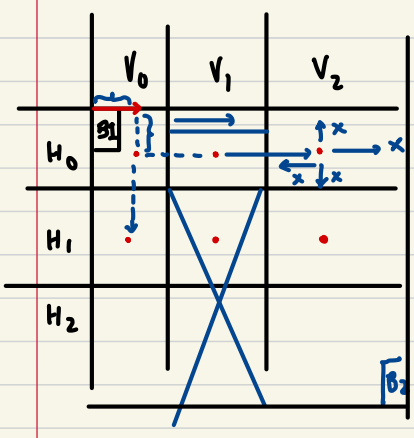
time complexity

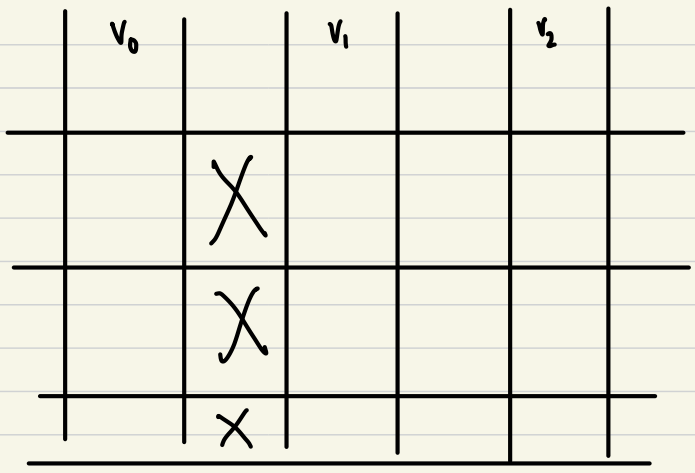
Step 1

Step 2 matrix → matrix → .... + n-1

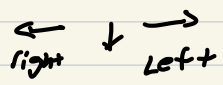
so a for loop

HW #4





Put it in a perspective of a person



$T(n)\ 9T(\frac{n}{3}) + n^3 + n^2 + n + 1$

$T(n) = 9^{\log_3 n}(1)T\ \frac{3n^3}{2}\left(\frac{1}{3^{\log_3 n}} - 1\right) + n\log_3 n + n\left(\frac{3^{\log_3 n} - 1}{2}\right) + \frac{9^{\log_3 n} - 1}{8}$

$t(n) = n^2 - \frac{3n^3}{2}\left(\frac{1}{n} - 1\right) + n\log_3 n + n\left(\frac{n-1}{2}\right)$

$\theta(n^3)$

HW # 2 pseudo code

```
function SortFile ( file-path , size-file , size buffer )
    I1 : of size   size buffer
    I2 :
    O1 :
    file = Load-file (file path)
    number of passes = Log2 (size of file) - 1
    number of files = Size of file / size of buffers
    first-pass (file , number of files)
    next-pass (1, number of passes , number files , size of file)
```

```
function FirstPass (file , number of files)
    While file is not empty {
      I1 = read(size of buffer)
      O1 = Quicksort (I1)
      O1 = flush ( tmpfile
```

```
function nextPass (currentpass , number-passes , number of files, size of file )
    If currentpass = number of pasres then return
    file Index = 0
    While true :
        file 1 = Load (currentpass, file Index - 1 )
        file 2 = Load (currentpass, file Index )
        While file 1 and file 2 are not empty
          I1  read (file 1)
          I2  read (file 2)
          O1  Merge (I1, I2)
        file Index = file Index + 2
        If file Index ≥ number of files  break
    nextpass (currentpass + 1 , number of passes, number files / 2 , size file * 2 )
```

$t(n) = T\left(\frac{n}{\frac{M}{B}}\right) + \frac{N}{B}\log_{\frac{M}{B}}\left(\frac{N}{B}\right)$

$t(n) = T\left(\frac{N}{2}\right) + \frac{N}{4}\log_2 \frac{N}{4}$

master theorem

$a = 1 \quad b = 2 \quad k = 1 \quad p = 1$

$\log_2 1 = 0 < k$

case #3 → 1

$p = 1$

$\theta(n^k \log^p n)$

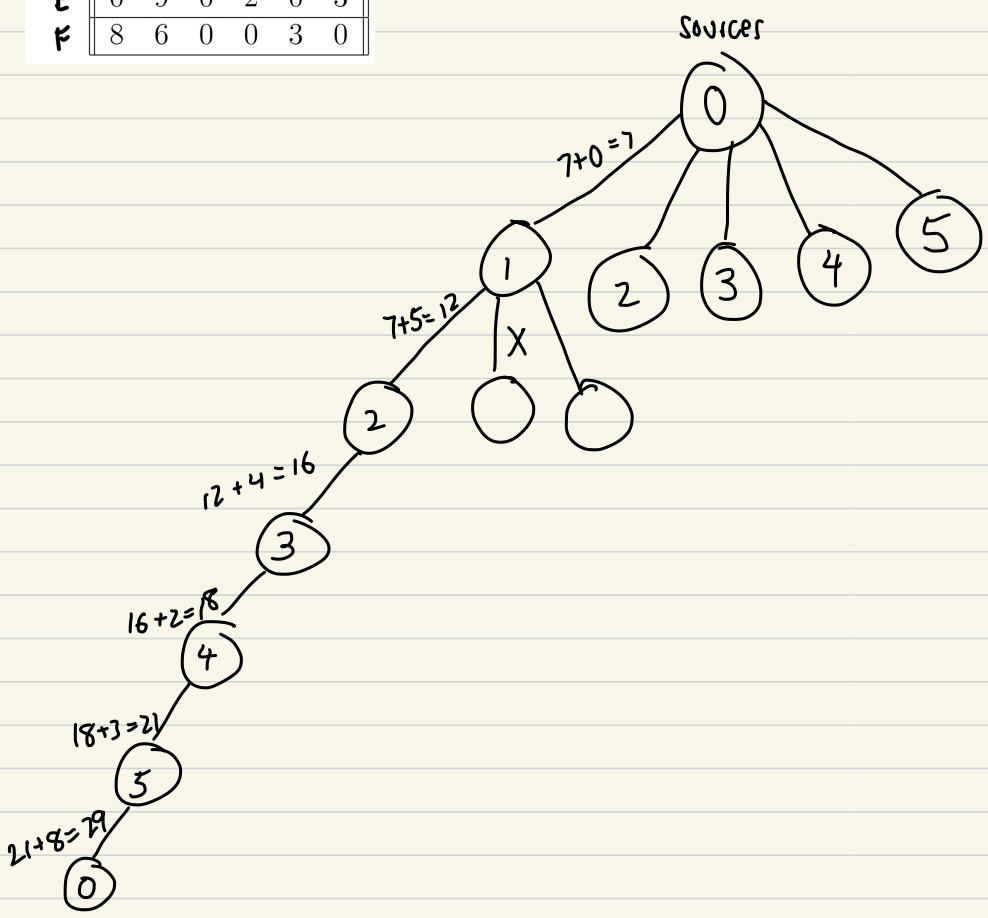
$\theta\left(\frac{N}{B}\log_{\frac{M}{B}}\left(\frac{N}{B}\right)\right)$

11/27/2023

Hw #3 Solutions

	A	B	C	D	E	F
A	0	7	1	0	0	8
B	7	0	5	0	9	6
C	1	5	0	4	0	0
D	0	0	4	0	2	0
E	0	9	0	2	0	3
F	8	6	0	0	3	0

step 1. Find all the hamiltonian cycles in the graph, and their distances
step 2. Find the minimums and maximums distances and their paths



1. Main method
2. Find distances (Recursive)
3. Find minimum distance
4. Find maximum distance

```
Find Distances ( Graph , numofNodes , currentNode , nodeCount , visitedNode , distance )

        If nodeCount == numofNodes and Graph [currentNode] [0] > 0 {
            distance = distance + Graph [currentNode] [0]
            distance . add (distance)
        }       Path . add(0)
        for each nextNode in Graph {
            If visited [nextNode] == False and Graph [current] [nextNode] >0 {
                    visited [nextNode] = True
                    findDistance (Graph , numofNodes, nextNode , nodeCount +1, VisitedNodes , distance + Graph [currentNode] [nextNode]
                      visited [nextNode] = False                                                                  Path . add (nextNode)
                    return
                    time complexity =

                    n!
```

distance [29, 23, 30, 23, 30, 29]
path [0, 1, 2, 3, 4, 5, 0]

no need to write psvedo code

Final review

1) floy'd algorithm
2) knapsack problem (DP or Greedy)
3) time complexity for a given psvedocode (iterative)
4) two problems : Asymptotic order
5) time complexity for a given psvedocode (recursive)
6) Dijkstra algorithm (true or false)
7) P=NP Give you a problem we covered in class
8) N- coloring problem
9) travelling salesperson problem tsp - decision , tsp search , tsp - opt
   - Tsp = Backtracking
   - Tsp = Branch & Bound
   - Tsp = Dp

$2+\left(\frac{n}{4}\right)+\log_2 n \in O(2^n)$

use master theorem

```
for(k=0, k<n; k++)  -> n
    for (p=1, p<(n-3), p++)  -> n
        for (j=2; j<p; j=j^2) -> Log Log n
            print(...)
```

not recursive
then drop : master theorem
Back substitution

$O(n^2 \log\log n)$

Why its $\log\log n$?

j	2^k	2^{2^k}
2	2^1	2^{2^0}
4	2^2	2^{2^1}
16	2^4	2^{2^2}
256	2^8	2^{2^3}
n		2^{2^k}

$n = 2^{2^k}$

$\log_2 n = 2k$

$\log\log n = k$

```
function f(n) {
    If n≤1  return 1   X
    for (i=0, i<(n-3), i++)  -> n
        for (j=n; j>1, j=j/2)  -> Log n -> 2^k = n/(2^k)
            Print(---)
```

master theorem

$f(\frac{n}{2})$ $f(\frac{n}{2})$ $f(\frac{n}{2})$ $f(\frac{n}{2})$ $\Big\rangle$ $4T(\frac{n}{2})$

$\Rightarrow 4T(\frac{n}{2}) + n\log n$

$a=4$ $p=1$

$b=2$

$k=1$

(H) $(n^{\log_b a})$

(H) (n^2)

$\log_2 4 = 2 > k$

W=10

Pt 2

i	Wi	Pi	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	20	40	0	0	0	0	0	0	0	0	0	0	0
2	6	24	0	0	0	0	0	0	24	24	24	24	24
3	3	6	0	0	0	6	6	6	24	24	24	30	30
4	7	40	0	0	0	6	6	6	24	40	40	40	(46)

$$M[i][w] = \max\left(\underbrace{M[i-1][w]}_{24},\ \underbrace{M[i-1][w-wi] + P[i]}_{6}\right)$$

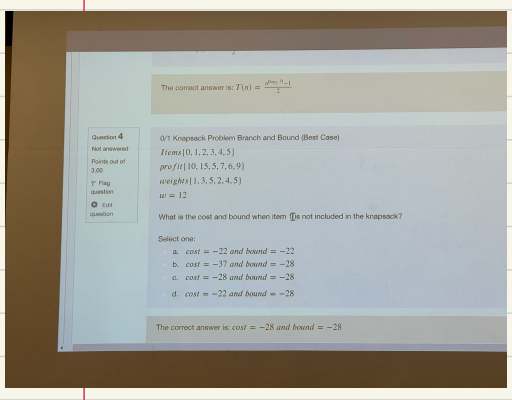
{∅} = 0

{1} = 0

{2} = 1

{3}

{ #4 , #3 , ∅ }



i 0 1 2 3 4 5

Pi 10, 15, 5, 7, 6, 9

Wi 1, 3, 5 2 4 5

1 + 0 + 5 + 2 + 4 + 0

cost = (10 + 0 + 5 + 7 + 6 + 9)(-1)

bound =

Cost = $\sum pi + \frac{Pi}{Wi}$

bound = $\sum pi$

1 + 3 + 5 + 11 + 1/4

Cost = (10 + 15 + 5 + 7 + $\frac{1}{4}$ (6))(-1)

bound = (10 + 15 + 5 + 7)(-1)