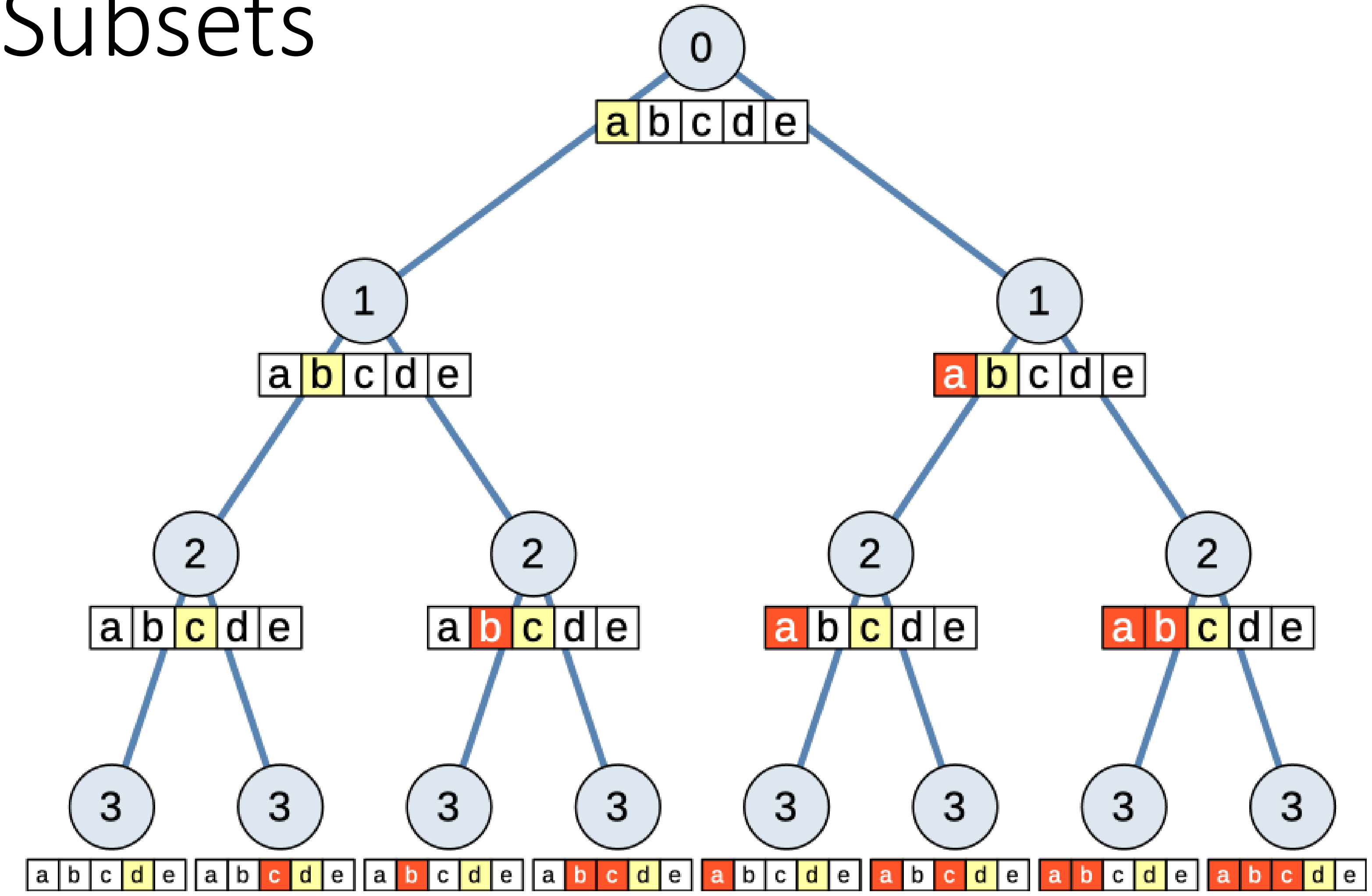
Competitive Programming

Search

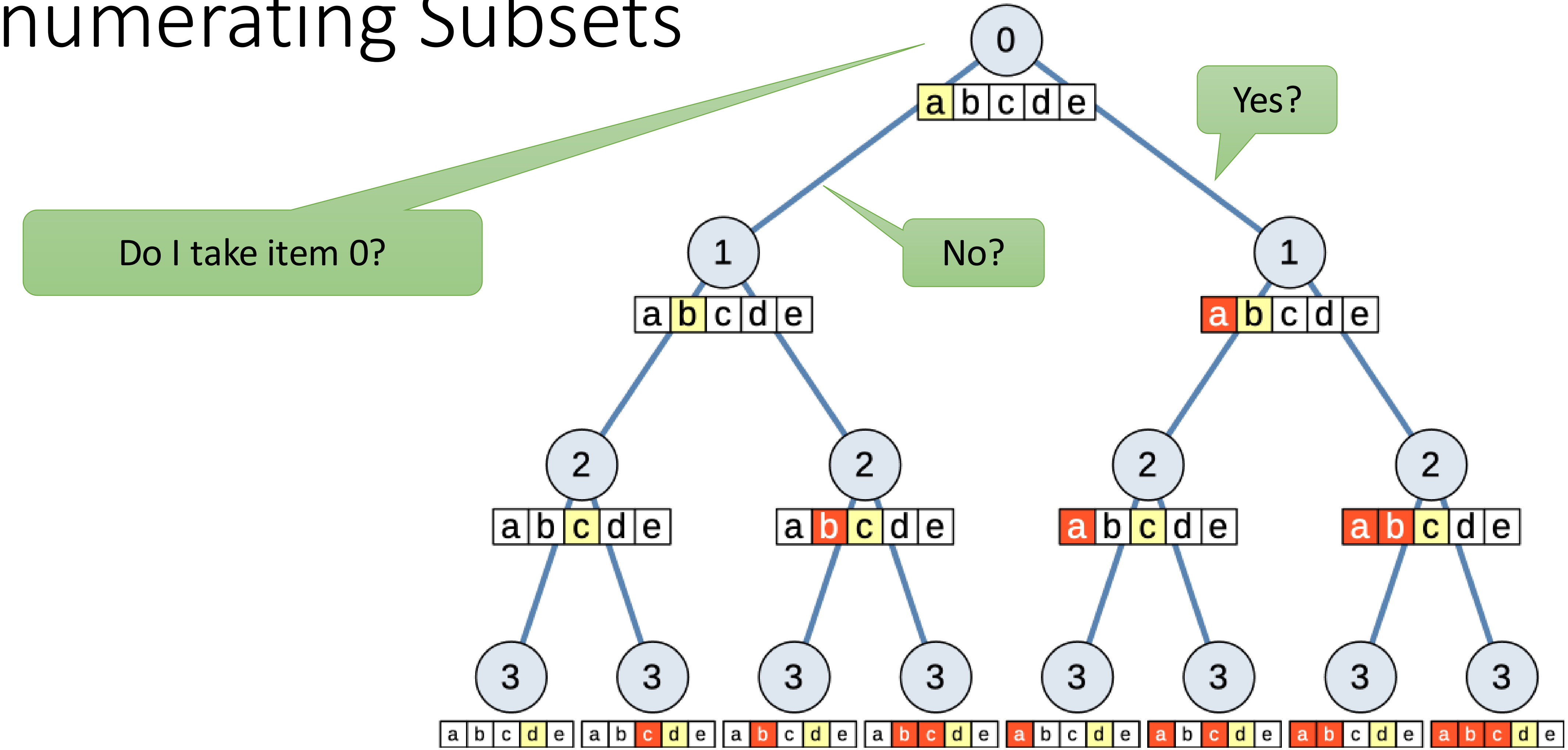
Search

- Considering possible solutions
  - All possible paths through a graph
  - All possible orderings of a sequence
  - All possible subsets of a set
  - All possible perfect matchings
  - …
- Like graph traversal
  - Implicitly defined graph
  - Typically, too large to be given in the input
  - Maybe too large to store in memory

Enumerating Subsets

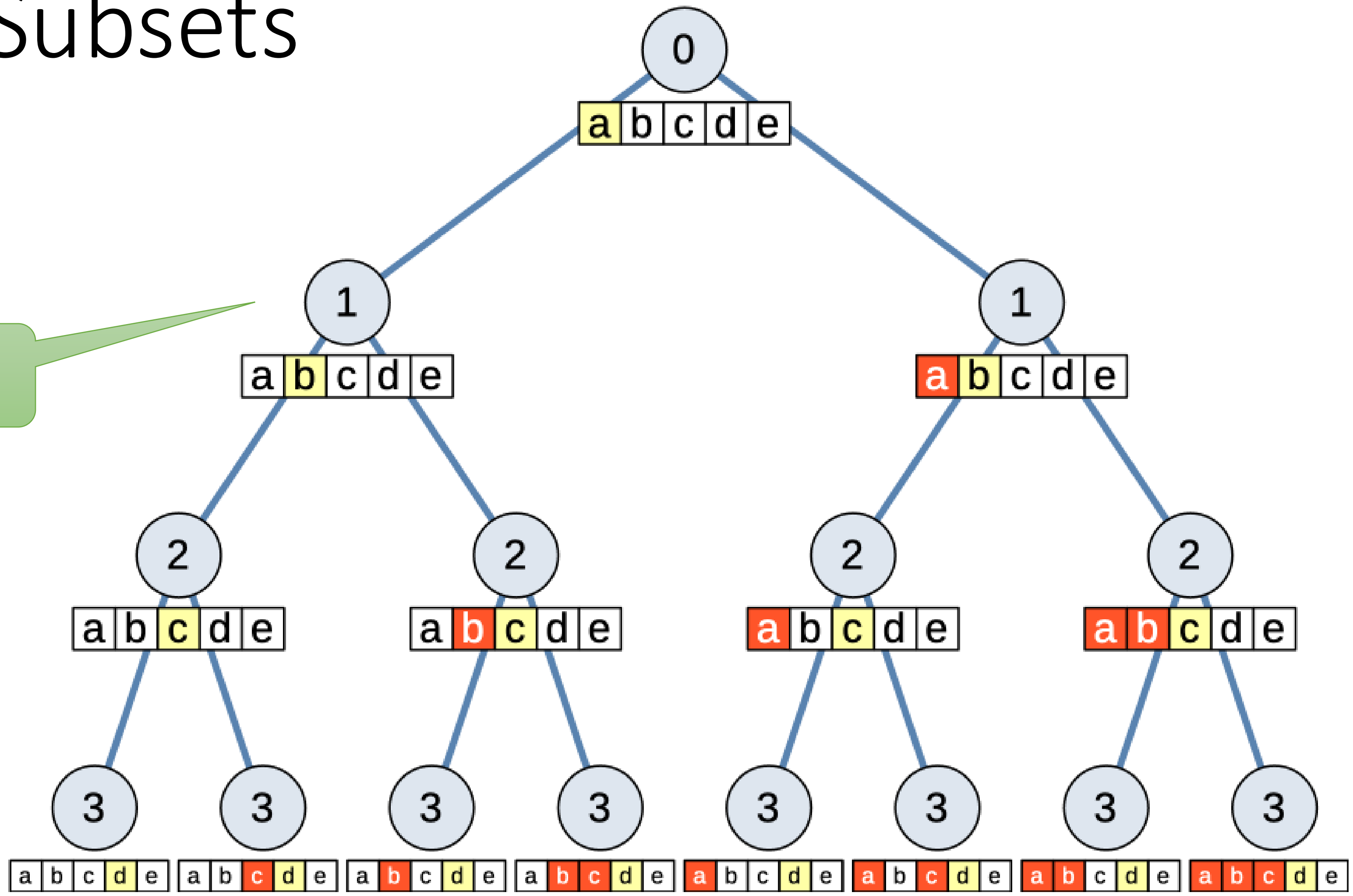


Enumerating Subsets



Enumerating Subsets

Do I take item 1?



Recursive Search

```
void search( int i, list<int> subset, list<int> seq ) {
  if ( i == n )
    // is this a solution?

  // Skip element i.
  search( i + 1, subset, seq );

  // Take element i.
  subset.add( seq[ i ] );
  search( i + 1, subset, seq );
  subset.pop_back();
}
```

Depth-First Traversal

```
void search( int i, list<int> subset, list<int> seq ) {
  if ( i == n )
    // is this a solution?

  // Skip element i.
  search( i + 1, subset, seq );

  // Take element i.
  subset.add( seq[ i ] );
  search( i + 1, subset, seq );
  subset.pop_back();
}
```

This is very easy to code.

Incremental (cheap) modification of the set.

Recursive Search

```
void search( int i, list<int> subset, list<int> seq ) {
  if ( i == n )
    // is this a solution?

  // Skip element i.
  search( i + 1, subset, seq );

  // Take element i.
  subset.add( seq[ i ] );
  search( i + 1, subset, seq );
  subset.pop_back();
}
```

Set maintained as a list of values.

Subset as a Boolean Array

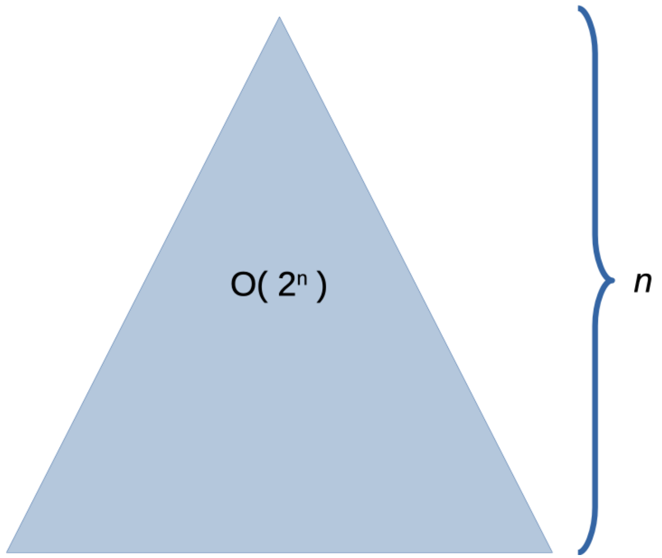
```
void search( int i, bool subset[ n ], list<int> seq ) {
  if ( i == n )
    // is this a solution?

  // Skip element i.
  search( i + 1, subset, seq );

  // Take element i.
  subset[ i ] = true;
  search( i + 1, subset, seq );
  subset[ i ] = false;
}
```

Subset as a Boolean Array

```
void search( int i, int bitmask, list<int> seq ) {
  if ( i == n )
    // is this a solution?

  // Skip element i.
  search( i + 1, bitmask, seq );

  // Take element i.
  search( i + 1, bitmask | ( 1 << i ), seq );
}
```

Search can be Expensive

- Search space is typically exponential
- Pruning can let us avoid subtrees that can't contain a solution.
- Branch-and-bound can let us skip subtrees that can't contain a solution.



Search Pruning

```
void search( int i, bool subset[ n ], list<int> seq ) {
  if ( i == n )
    // is this a solution?

  if ( ! validSolution( subset ) )
    return;

  // Skip element i.
  search( i + 1, subset, seq );

  // Take element i.
  subset[ i ] = true;
  search( i + 1, subset, seq );
  subset[ i ] = false;
}
```

Do prior choices violate the problem constraints?

Branch-and-bound

```
void search( int i, int cost, bool subset[ n ], list<int> seq ) {
  if ( i == n )
    // is this a solution?

  if ( cost >= bestSolution )
    return;

  // Skip element i.
  search( i + 1, subset, seq );

  // Take element i.
  subset[ i ] = true;
  search( i + 1, subset, seq );
  subset[ i ] = false;
}
```

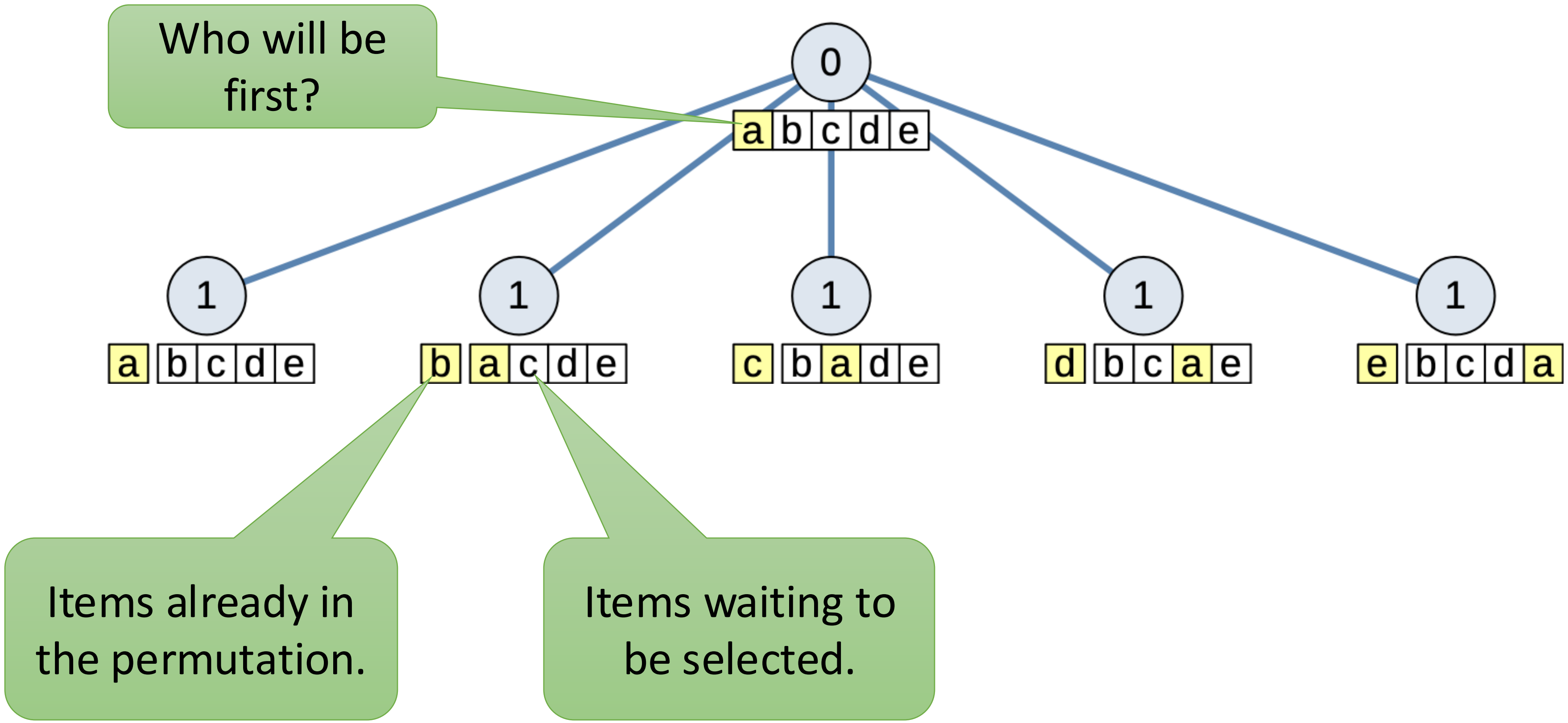
Is this solution already too expensive?

Branch-and-bound

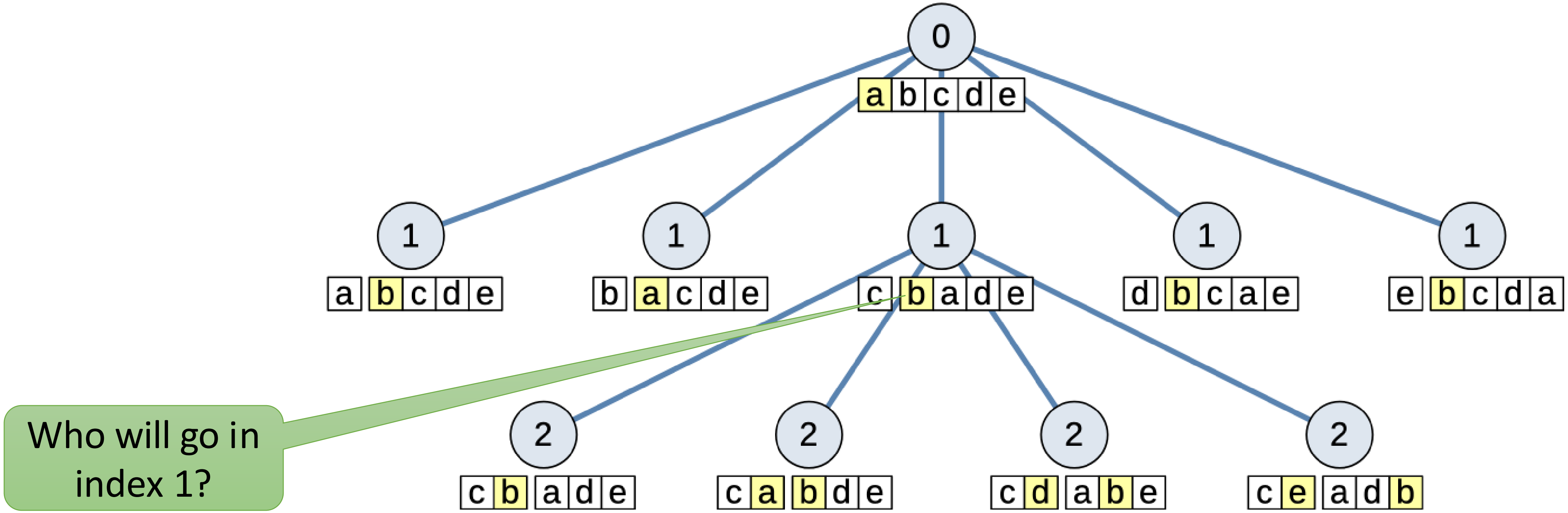
```
void search( int i, int cost, bool subset[ n ], list<int> seq ) {
  if ( i == n )
    // is this a solution?

  if ( cost + remainingCostBound( subset ) >= bestSolution )
    return;

  // Skip element i.
  search( i + 1, subset, seq );

  // Take element i.
  subset[ i ] = true;
  search( i + 1, subset, seq );
  subset[ i ] = false;
}
```

A bound on remaining cost can provide more pruning

Enumerating Permutations



Enumerating Permutations



Branch-and-bound

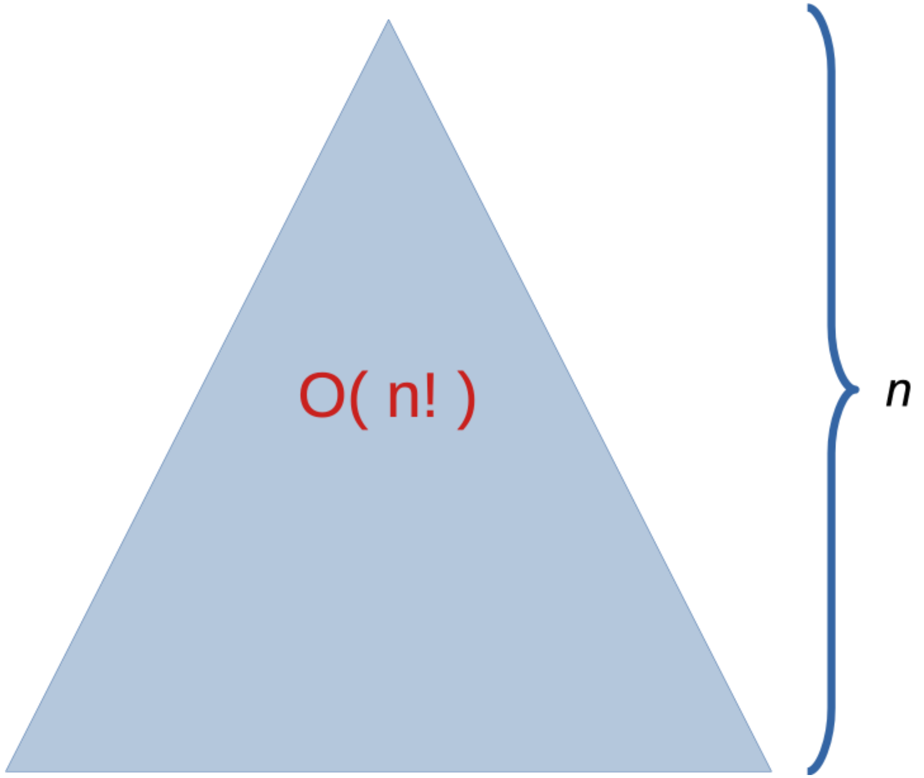
```
void search( int i, list<int> perm ) {
  if ( i == n )
    // is this a solution?

  for ( int j = i; j < n; j++ ) {
    swap( perm[ i ], perm[ j ] );
    search( i + 1, perm );
    swap( perm[ i ], perm[ j ] );
  }
}
```

Permutation is maintained in place.

Put the sequence back like it was.

Expensive!

- All possible permutations is a larger search space.
- Pruning can be even more important.

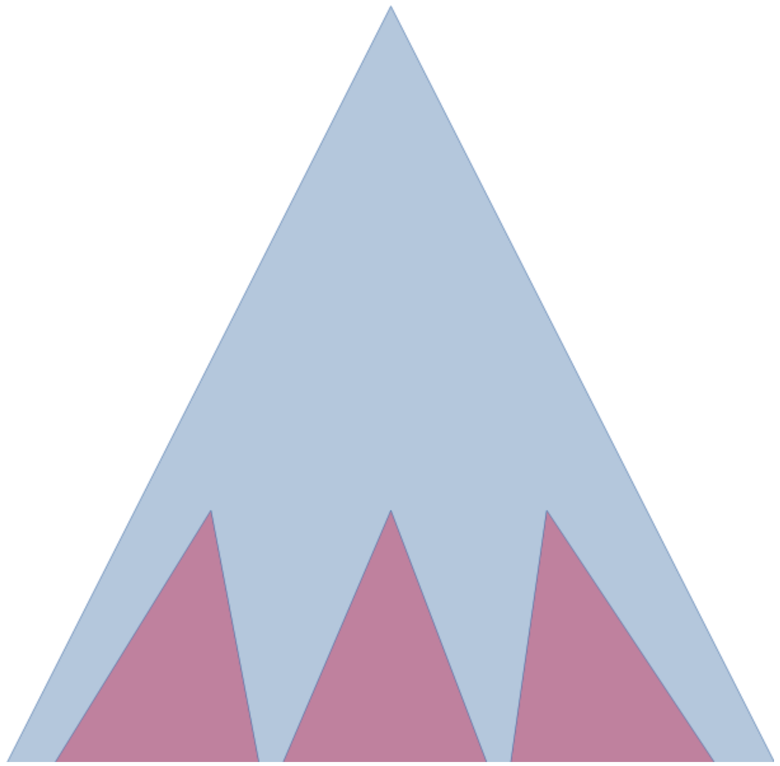


Search Pruning

```
void search( int i, list<int> perm ) {
  if ( i == n )
    // is this a solution?

  if ( perm[ 0 ] … perm[ i – 1 ] can’t be a solution )
     return;

  for ( int j = i; j < n; j++ ) {
    swap( perm[ i ], perm[ j ] );
    search( i + 1, perm );
    swap( perm[ i ], perm[ j ] );
  }
}
```

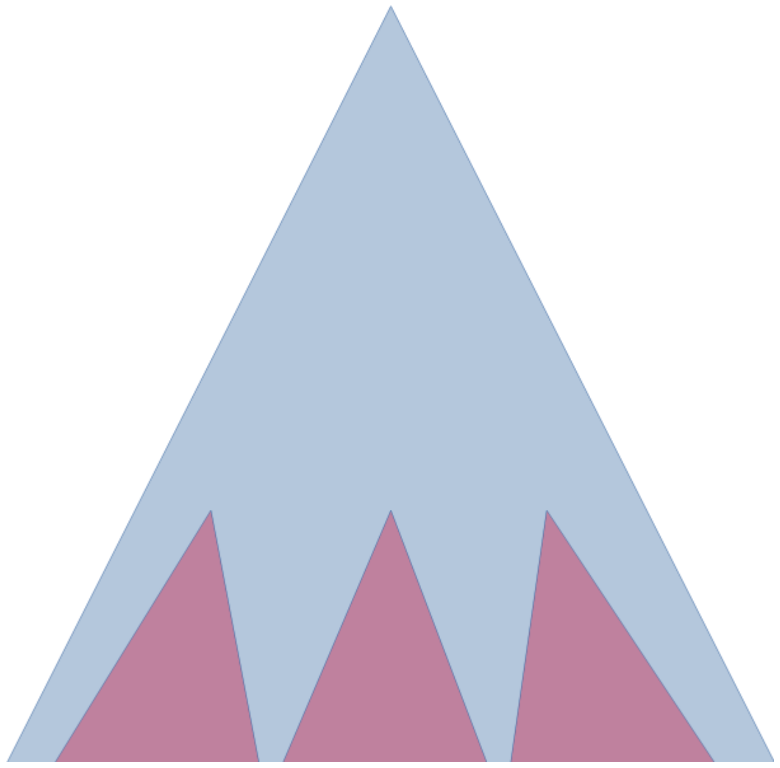
Duplicate Subtrees

- The same subtree can show up more multiple times.
- Caching solutions can avoid duplicate work.
- How do you make a cache for a subset of elements?



Duplicate Subtrees

- The same subtree can show up more multiple times.
- Caching solutions can avoid duplicate work.
- How do you make a cache for a subset of elements?
  - Easy. Use a bit mask as the index.



Subtree Caching

```
void search( int i, int cost, int remaining, list<int> perm ) {
  if ( i == n )
    // is this a solution?

  if ( cache[ remaining ] &&
       cost + cache[ remaining ] is too high )
      return;

  for ( int j = i; j < n; j++ ) {
    swap( perm[ i ], perm[ j ] );
    search( i + 1, cost + ?, remaining ^ perm[ i ], perm );
    swap( perm[ i ], perm[ j ] );
  }

  cache[ remaining ] = ?;
}
```