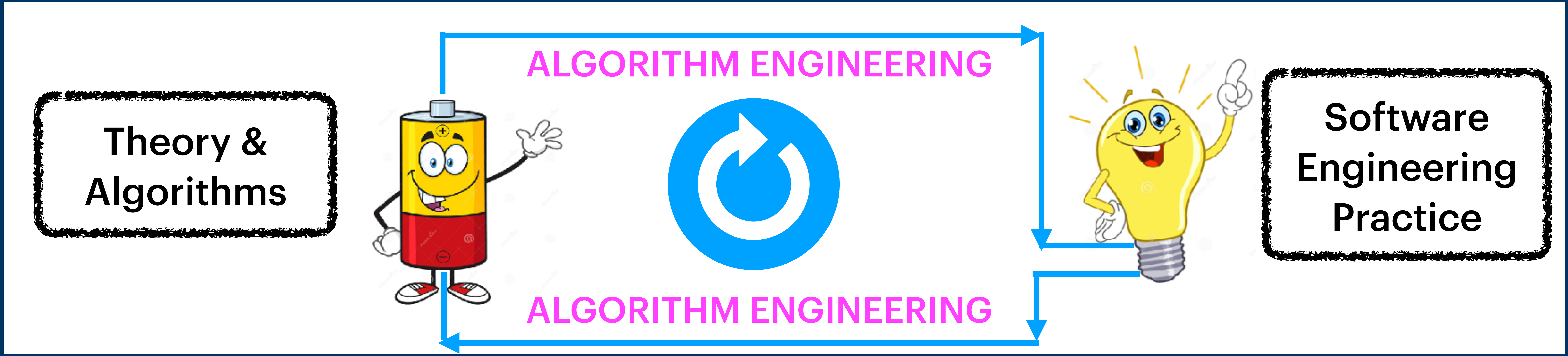


ALGORITHM ENGINEERING

Lecture 6:
Implementation Phase - 1:

M. Oğuzhan Külekci - kulekci@itu.edu.tr

How to make it run faster ?

The central question in algorithm engineering

- Every step in the design of a solution has an effect on speed.
- Algorithm design, analysis, and a basic implementation are done, and we want to improve that implementation.

	$N = 10$	13	14	20	27	30
V1	69.68					
V2 (a)	6.97					
V3 (c)	2.81					
V4 (c)	.57	13.71	74.86			
V5 (a)	.10	.08	.43	49.52		
V6 (a)		.02	.07	2.61	92.85	
V8 (a)		.01	.04	1.09	60.42	137.92

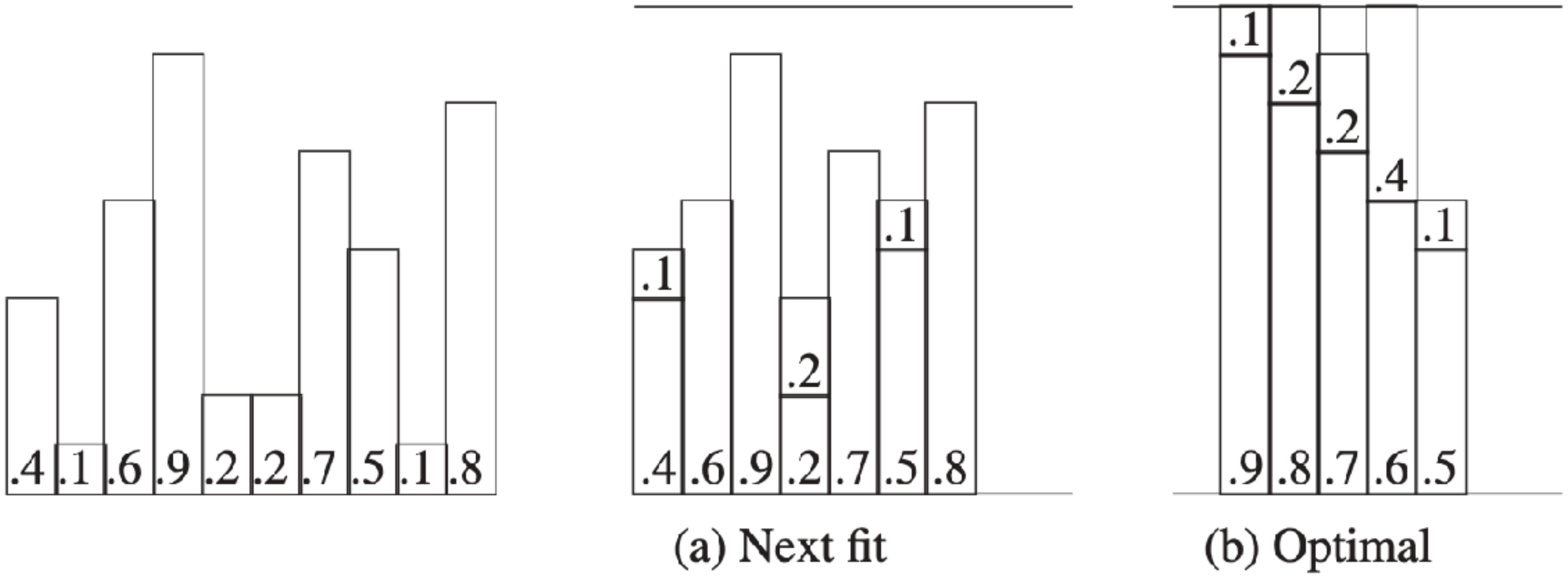
Reduce Either

Instruction counts OR **Instruction times**

(Algorithm Tuning) (Code Tuning)

The case study

Bin packing with next fit

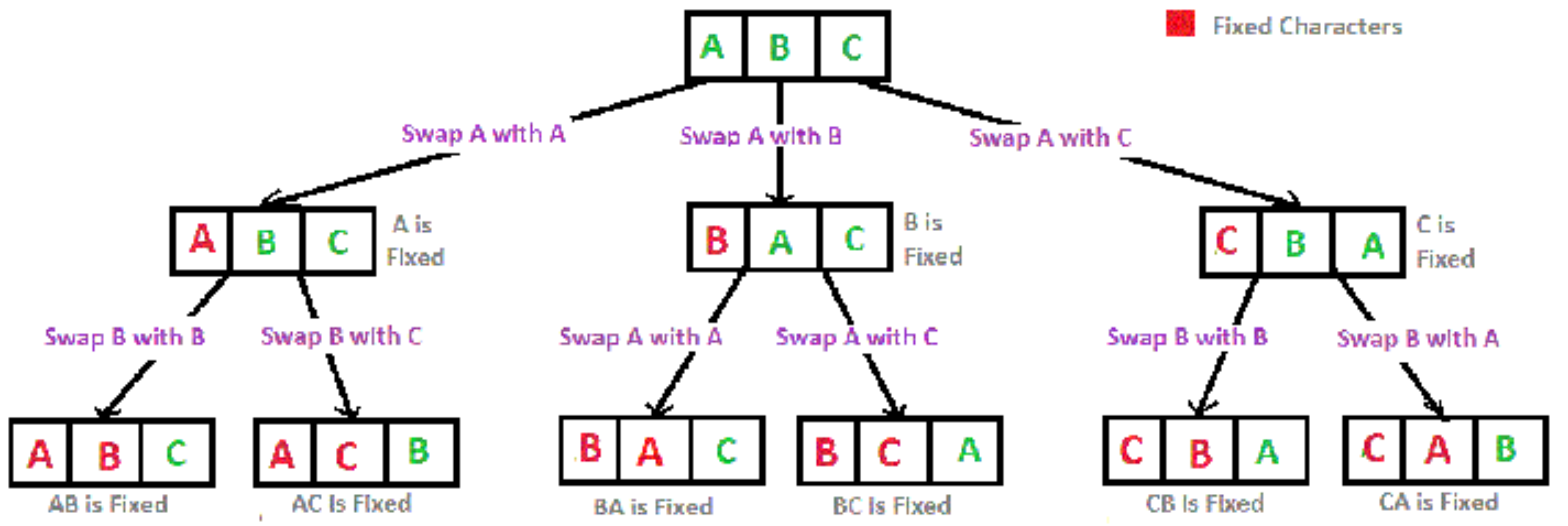



```
global    list[0..n-1];    // list to be packed
global    optcost;         // minimum bin count

procedure binPack (k) {
  if (k == n){
      b = binCount();              // use next fit
      if (b < optcost) optcost = b;
  }
  else
    for (i = k; i < n ; i++) {
        swap (list, k, i);         // try it
        binPack (k+1);             // recur
        swap (list, k, i);         // restore it
    }
}
```

Exhaustive Search with binPack(0)

$$O(n \cdot n!)$$



Recursion Tree for Permutations of String "ABC"

Generating all permutations of an array is an Interesting topic.

Branch & Bound

Bin packing with next fit

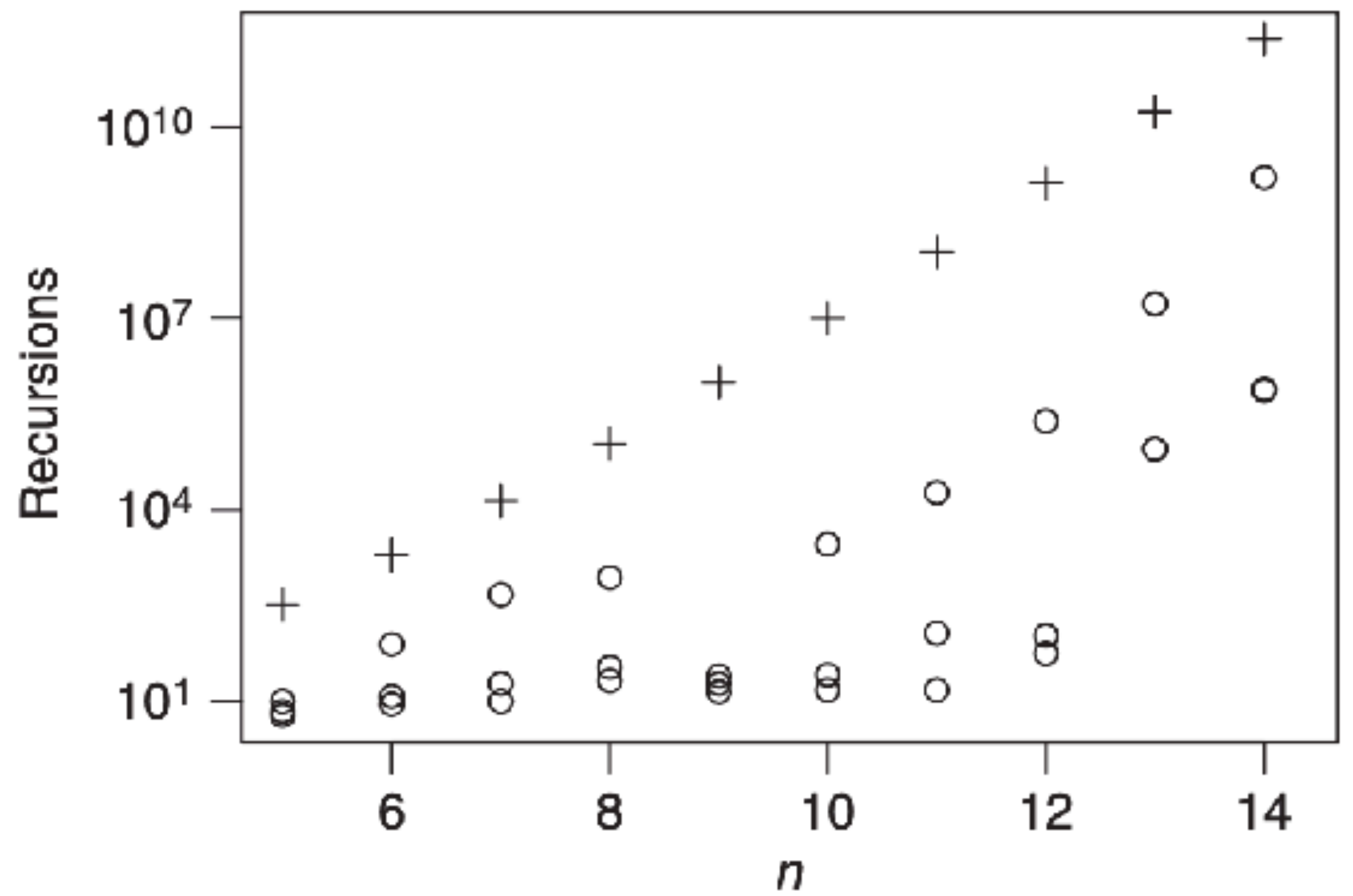
Find a condition to limit further investigation on an exhaustive search procedure.

0	k	k+1	n-1
list[0..k]		list[k+1..n-1]	
binPack(k)		$lowerBound = \lceil a[k+1] + a[k+2] + .. + a[n-1] - (1 - a[k]) \rceil$	

If binPack(k) + lowerBound $\geq$ optCost, then no need to try the rest of the permutations

```
for (i = k; i<n ; i++) {
   swap(list, k, i)         // try it
   b = binCount(k);
   w = weightSum(k+1);
   if(b+ Ceiling(w-(1-list[k])) < optcost)
          binPack (k+1);    // recur if needed
   swap (list, k, i)        // restore it
```

O(n) , improve ?



Propagation

While computing each recursion, instead of a full execution, try to propagate the previous results to reduce the computation time. Hence, enrich the binPack(k) with more parameters

```
binPack (k, bcount, capacity, sumwt) {
if (k == n) {
 if (bcount < optcost)
    optcost=bcount;
}
else {
  for (i=k; i<n; i++ ) {
     swap (list, k, i);                    // try it
     if (capacity + list[k] > 1) {   // does it fit?
         b = bcount + 1;               // use new bin
         c = 1 - list[k];
     }
     else {
         b  = bcount;                  // use old bin
         c  = c - list[k];
     }
     w = sumwt - list[k];              // update sumwt
     if (b+Ceiling(w-c) < optcost)   // check bound
          binpack(k+1, b, c, w);       // recur if necessary
     swap (list, k, i);                // restore it
  }
}
}
```

- sumwt: Sum of the all items in the current list
- bcount : number of bins used
- capacity:

- **Careful analysis of the current item helps to improve our lower bound computation**
- **While computing the lower bound, instead of summing all remaining items, which brings O(n) load, introduce sumwt to make this O(1).**

Preprocessing

Do some calculation before running the algorithm to improve the performance.

- What if we have a good ***optcost*** at the beginning in. nextFit bin packing ?
- Having such a good ***optcost*** at the beginning will save a lot of recursions.
- How to find such a good initial ***optcost*** ?

First fit decrasing (FFT) heuristic:

- Sort the items in decreasing order.
- Run first fit bin packing.
- Use the result as the initial ***optcost ?***

Essential Edges in a Graph

A new case study

```
procedure findEssential (G) constructs S
     S.initialize(n)    //  Subgraph initially empty
     P.initialize(G)     //  Priority queue of edges of G

     while (P.notEmpty() ){
        <x, y, cost> = P.extractMin();
        if (S.distance(x,y) > cost) S.insert(x,y,cost);
   }
```

Distance is by Dijkstra's shortest path algorithm

```
procedure S.distance(s, d) returns distance from s to d
   For all vertices v: v.status = unseen;
   Ps.init(s,0);                  // insert s with distance 0
   s.status = inqueue;
   while (Ps.notEmpty()) {
      <w, w.dist> = Ps.extractMin();
      w.status = done;
      if (w == d) return w.dist;          // found d
      for (each neighbor z of w) {
         znewdist  = w.dist + cost(w,z);  // relax
         if (z.status == unseen)
            Ps.insert(z, znewdist);
         else if (z.status == inqueue)
            Ps.decreaseKey(z, znewdist);
    }//while
    return   +Infinity;                // didn't find d
```

In a given graph G(n,m), if an edge e between the vertices x and y has the unique least-cost from x to y, then e is an essential edge.

- Number of vertices in G is n
- Number of edges in G is m
- Number of edges in S, the output, is m'

Why we seek shortest-path in S rather than G: Edge e can be identified as essential or nonessential by considering **only paths of essential edges** with costs smaller than e.

Memoization & Finessing

Memoization: Store the results of previous calculations not to repeat them again later.

Store the shortest path from vertex x to y on S in a matrix D for all x and y.

Finessing: Avoid an expensive calculation via a cheap approximation that does not change the result.

Between any two nodes, if the stored distance on S is less than the edge between them, then no need to execute the distance function since it cannot update the result.

Runtime	$n = 800$	$n = 1000$	$n = 1200$	$n = 1400$
v_0	24.11	49.04	87.57	144.97
v_1	.49	.83	1.32	1.91
v_0/v_1	49.20	48.24	66.34	75.90

```
procedure S.distance(s, d, ecost) returns
        distance from s to d, or upper bound D[s,d]

if (D[s,d] <= ecost) return D[s,d];
For all vertices v: v.status = unseen;
Ps.init(s,0)                    // insert s with distance 0
s.status = inqueue;
while (Ps.notEmpty()) {
  <w, w.dist> = Ps.extractMin();
  if (w.dist < D[s,w]) D[s,w] = w.dist;
  w.status = done;
```

Loop abort

Find ways to break loop execution early, similar to branch-and-bound.

```
bound = (global) bound on max essential edge cost

procedure findEssential (G) constructs S
     S.initialize(n) // Subgraph initially empty
     P.initialize(G) // Priority queue of edges of G
     while (P.notEmpty()){
        <x, y, cost> = P.extractMin();
        if (S.distance(x,y) < cost ) S.insert(x,y,cost);
        if (cost > bound) break;       // loop abort
     }
```

If we know the weight of the maximum essential edge, we can stop the while loop earlier.

The question is how to find such a bound !

Twice (!) the largest distance between any two nıdes on S is such a bound. This can be computed tightly with Dijkstra or loosely with BFS.

Read discussions in the chapter to understand why BFS is a better choice then Dijkstra and also the fine tuning.

We are adding an expensive calculation to compute the maximum essential edge weight ! However, this will pay off its cost by cutting the need to repeat the cheaper one many times. Experiments showed more than 90% cut-off.

Customizing the data structure

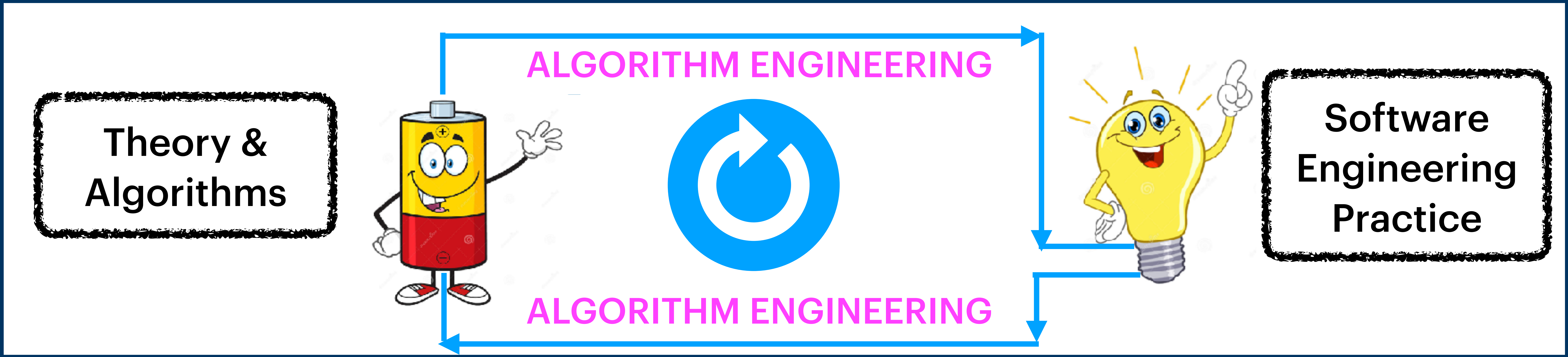
- We use some standard data structures in our programs.
- Profile which operations of the used data structure is heavily used.
- Look for finding ways to make them run faster by customizing the data structure

For instance, the priority queue data structure is implemented in essential edge detection problem.Profiling the execution it is observed that extractMin operation is heavily used.
Read the chapter to get how the heap is customized ….

- We have focused on reducing the instruction count by algorithm tuning.
- Next week we will continue
- Read chapter 4 until 4.1.1 on page 116
- Here is the HW for next week

1. Suppose you can improve the running time of a given program by a factor of 2 in one day's work, but no more than a factor of 32 (five day's work) can be squeezed out of any given program. Your time is worth $100 per hour. This includes time waiting for a program to finish a computation. Which of the following scenarios is worth the price of a week of algorithm engineering effort?
   a. The program is executed once a day and takes one hour to run.
   b. The program is executed a million times per day, and each run takes one second.
   c. The program is executed once a month and takes one day to run.

NEXT LECTURE ...



ALGORITHM ENGINEERING

Lecture 6:
Implementation Phase - II

M. Oğuzhan Külekci - kulekci@itu.edu.tr