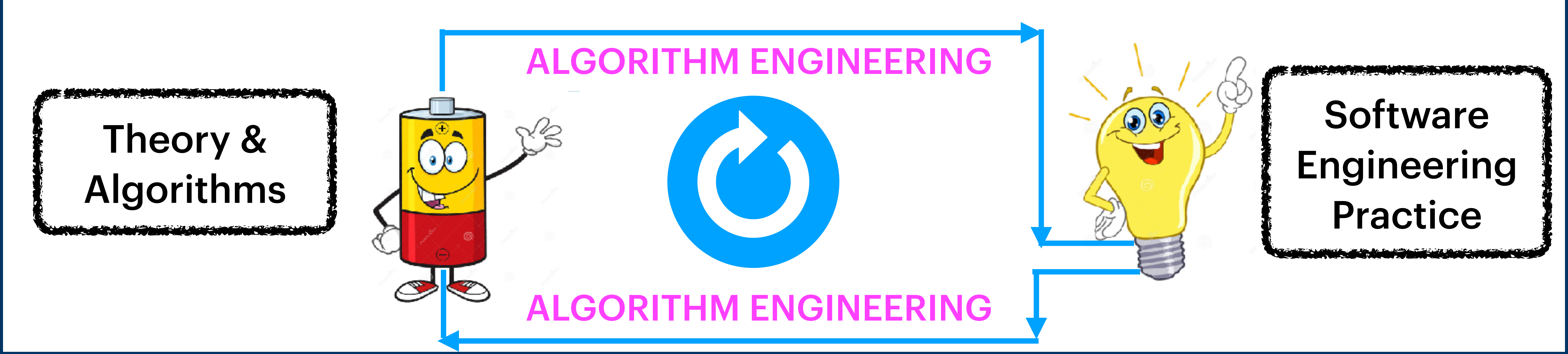


ALGORITHM ENGINEERING

Lecture 6:
Implementation Phase - 2:

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.

Reduce Either

Instruction counts	OR	Instruction times
(Algorithm Tuning)		(Code Tuning)

Recursion-Heavy Algorithms

1. **Exhaustive enumeration based algorithms:** Produce all possible cases, find the best solution. *Example: Next-fit approach in bin packing problem*

2. **Divide-and-conquer type algorithms:** Split the problem into smaller-size problems, that are easier to handle. *Example: Quick - sort*

```
Quicksort (A , lo, hi )
   if (lo >= hi ) return;       // Cutoff test
   p = A[lo]                    // Partition element p
   x = Partition(A, lo, hi, p)  // Partition around p
   Quicksort (A, lo, x-1)       // Recur left
   Quicksort (A, x+1, hi)       // Recur right
```

Such algorithms include many recursive calls. The way to speed them up is to **skip** some of these recursive callas

Recursion-Heavy Algorithms

Pruning

Algorithm Pruning: Insert simple tests to prune recursive calls. Boost the strength of these tests by using preprocessing or by changing computation order

1. **Backtracking:** While traversing over the exhaustively enumerated solutions, abandon the generation, stop investigation when you understand it will not help.

2. **Branch-and-bound:** Include some tests to see whether the result can improve. If not, skip this trial. Can be boosted via **preprocessing** and similar efforts.

We covered the case for exhaustive enumeration in the bin packing.

Let us discuss the case for divide-and-conquer type.

Modify quick sort to return k-th smallest (largest) search, all items larger (smaller) than an element

Recursion-Heavy Algorithms

Controlling the sub-problem size

Remove elements from subproblems before recurring; add or subtract work to balance subproblems.

Example 1 : In next-fit bin packing, combine the items that sum up to exactly 1.

Example 2 : In quick-sort, avoid choosing unbalanced partitions.

Notice that both will introduce extra computation, but this will pay off its price…

Recursion-Heavy Algorithms

Shrink cost per stage

Try speeding up each recursion.

Example, the propagation technique in bin packing

In divide-and-conquer type algorithms, consider self-tuning the parameters, e.g., choose pivot among sampling .1% of the space.

Yet another point maybe to hybridize, during the quick sort, switch to insertion sort when the partition is small.

Iterative Algorithms

Shrink cost per stage

Dynamic Programming: Iteratively fills an array

Greedy Algorithms: Construct a solution by iteratively selecting an item from a priority queue

On such solution approaches,
the key is to speed up the data structure used, and also the memory utilization

Memoization, loop abort, filtering …

Tuning the code

Loop

```
a[0..n-1] contains elements to be sorted

for (i = 1; i<n; i++ ) {
    // Invariant:  a[0..i-1] is sorted
    // Invariant:  a[i..n-1] not yet sorted
    for (j=i;  j>0 && a[j]>a[j-1]  ; j--) {
        //Invariant: a[j] > a[j-1]
        tmp    = a[j];
        a[j]   = a[j-1];
        a[j-1] = tmp;
    }
}
```

```
a[0..n-1] contains elements to be sorted

for (i = 1; i < n; i++ ) {
    // Invariant: a[0..i-1] is sorted
    // Invariant: a[i..n-1] not yet sorted
    int tmp = a[i];
    for (j=i; (j>0 && a[j]>tmp); j--) {
      // Invariant: hole is at a[j]
      a[j] = a[j-1];
    }
    a[j] = tmp;
}
```

Code motion out of loops: Remove unnecessary operations out of the loops

Compiler optimizers usually do this, but sometimes they may need help

Tuning the code

Sentinels

```
a[0..n-1] contains elements to be sorted

for (i = 1; i < n; i++ ) {
    // Invariant: a[0..i-1] is sorted
    // Invariant: a[i..n-1] not yet sorted
    int tmp = a[i];
    for (j=i; (j>0 && a[j]>tmp); j--) {
      // Invariant: hole is at a[j]
      a[j] = a[j-1];
    }
    a[j] = tmp;
}
```

	$n = 40{,}000$	80,000	160,000
Original	0.65	2.66	10.59
With code motion	0.40	1.61	6.43
With sentinel	0.67	2.72	10.91
With motion + sentinel	0.30	1.22	4.88

```
a[1..n] contains elements to be sorted
a[0]    contains the sentinel value -Infinity

for (i = 1; i <= n; i++ ) {
    // Invariant: a[1..i-1] is sorted
    // Invariant: a[i..n] not yet sorted
    int tmp = a[i];
    for (j = i; a[j]>tmp; j--) { //new test
      // Invariant: hole is at a[j]
      a[j] = a[j-1];
    }
    if (j==0) a[j+1] = tmp;
    else a[j] = tmp;
}
```

Tuning the code

Procedures

Procedure A calls procedure B, then what happens

- allocate space on call stack for the parameters of B
- initialize all variables of B
- save the state (registers) of A to reconstruct them on the return

  Such operations create an overhead that we want to avoid.

Tuning the code

Procedures - Inlining

```
perms (double*  a, int k, int n) {
  int i;
  if (k == n-1) {
     printPerm(a, n);
     return;
  }
  for (i=k; i < n; i++) {
     swap (a, k, i);      // try it
     perms(a, k+1, n);    // recur
     swap (a, k, i);      // restore it
   }
}
```

```
tmp = a[k];
a[k] = a[i];
a[i] = tmp;
```

Inlining: Replace procedure calls with proper code expansions

Tuning the code

Procedures - Collapse Procedure Hierarchies

```
perms (double*  a, int k, int n) {
  int i;
  if (k == n-1) {
     printPerm(a, n);
     return;
  }
  for (i=k; i < n; i++) {
     swap (a, k, i);      // try it
     perms(a, k+1, n);    // recur
     swap (a, k, i);      // restore it
   }
}
```

```
perms (double*  a, int k, int n) {
    int i, j;
    double tmp;
    if (k == n-1) {
      printPerm(a, n);
      return;
    }
    if (k == n-2) {
       printPerm(a, n);
       tmp = a[k]; a[k] = a[k+1]; a[k+1] = tmp;
       printPerm(a, n);
       tmp = a[k]; a[k] = a[k+1]; a[k+1] = tmp;
       return;
    }
    for (i=k; i < n; i++) {
      tmp = a[k]; a[k] = a[i]; a[i] = tmp; //swap k
        for (j = k+1; j < n; j++) {
         tmp=a[k+1]; a[k+1] = a[j]; a[j]=tmp; //swap k+1
              perms(a, k+2, n);         // recur
         tmp=a[k+1]; a[k+1] = a[j]; a[j]=tmp; //restore k+1
          }
      tmp=a[k]; a[k]=a[i]; a[i]=tmp; //restore k
    }
 }
```

Swaps are replaced with inline code. Single for loop replaced with nested double for loop to reduce the number of procedure calls.

Tuning the code

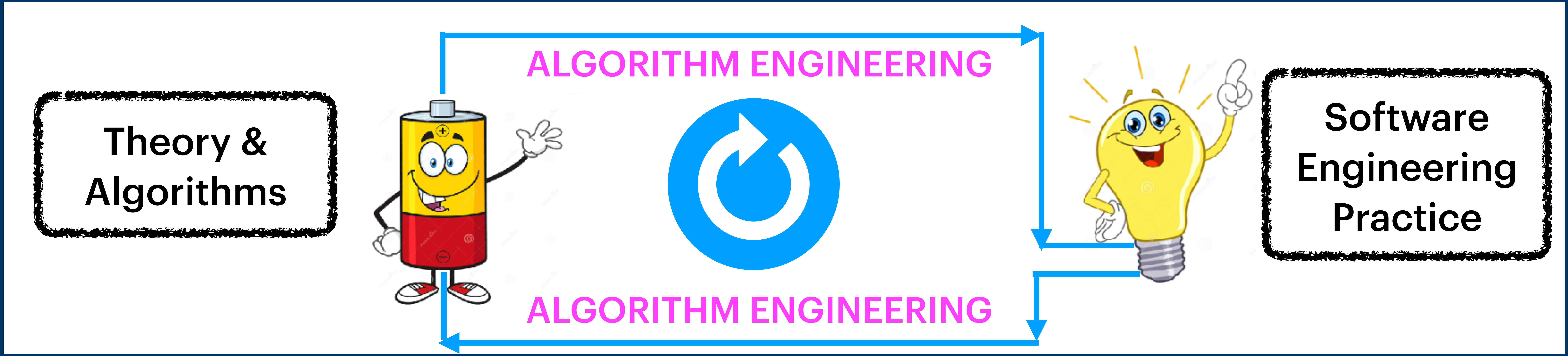
Compiler optimization usually does a better job

	$n = 11$	12	13
Original	1.44	17.29	224.73
With inline	1.59	19.06	247.76
With collapse	1.29	9.39	199.52
With collapse + inline	0.95	6.56	148.29

	$n = 11$	12	13
Original	0.43	6.53	65.03
With swap inline	0.44	6.75	65.96
With collapse	0.45	2.54	82.44
With inline + collapse	0.47	2.77	70.94

with -O3

NEXT LECTURE ...



ALGORITHM ENGINEERING

Lecture 6:
Implementation Phase - 3

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