Georgia Tech

CSE 6140/ CX 4140:

Computational Science and Engineering ALGORITHMS

Instructor: Anne Benoit

Visiting Associate Professor, CSE

Based on slides by Bistra Dilkina

Greedy Algorithms to compute Minimum Spanning Trees (MST)

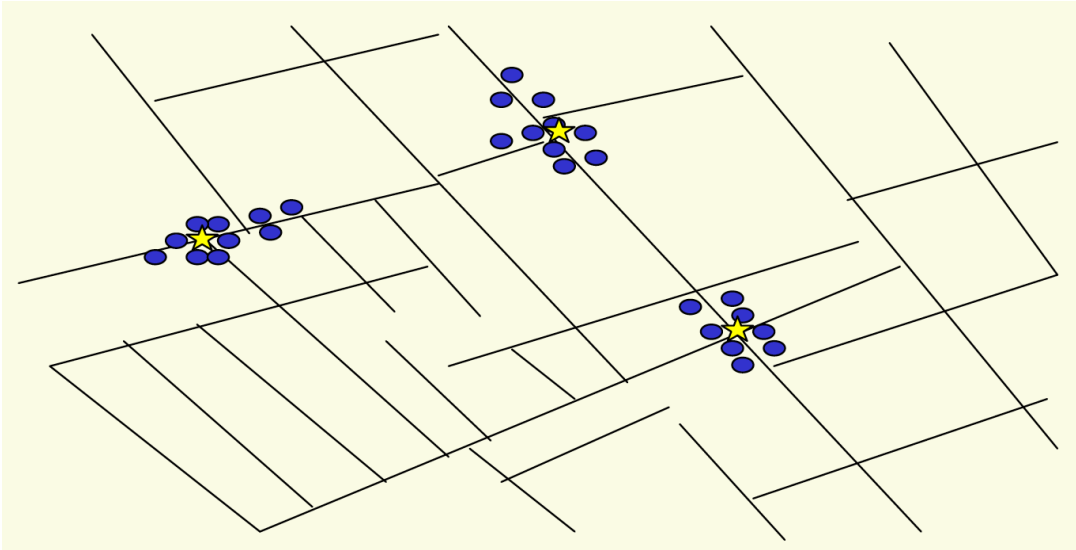
Kruskal's algorithm. Start with $T = \phi$. Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle.

Reverse-Delete algorithm. Start with T = E. Consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T.

Prim's algorithm. Start with some root node s and greedily grow a tree T from s outward. At each step, add the cheapest edge e to T that has exactly one endpoint in T.

All three algorithms produce an MST.

KT4.7 Clustering



Outbreak of cholera deaths in London in 1850s.
Reference: Nina Mishra, HP Labs

Clustering

Clustering. Given a set U of n objects labeled $p_1, \ldots, p_n$, classify into coherent groups.

photos, documents. micro-organisms

Distance function. Numeric value specifying "closeness" of two objects.

number of corresponding pixels whose intensities differ by some threshold

Fundamental problem. Divide into clusters so that points in different clusters are far apart.

- Routing in mobile ad hoc networks.
- Identify patterns in gene expression.
- Document categorization for web search.
- Similarity searching in medical image databases
- Skycat: cluster 10^9 sky objects into stars, quasars, galaxies.

Clustering of Maximum Spacing

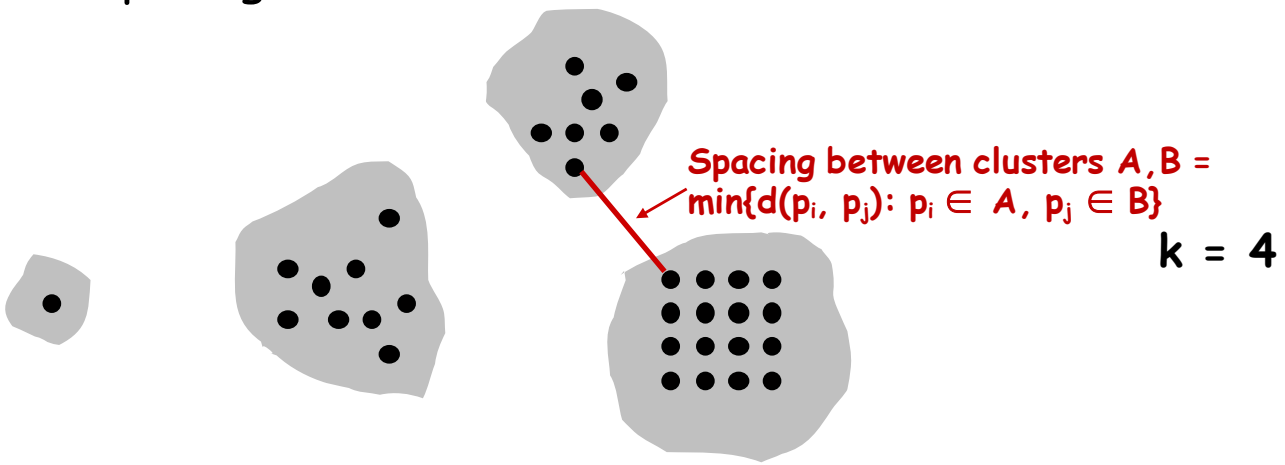
k-clustering. Divide objects into k non-empty groups.

Distance function. Assume it satisfies several natural properties.

- $d(p_i, p_j) = 0$ iff $p_i = p_j$ (identity of indiscernibles)
- $d(p_i, p_j) \geq 0$ (nonnegativity)
- $d(p_i, p_j) = d(p_j, p_i)$ (symmetry)

Spacing. Min distance between any pair of points in different clusters.

Clustering of maximum spacing. Given an integer k, find a k-clustering of maximum spacing.



Greedy Clustering Algorithm

Single-link k-clustering algorithm.

- Form a graph on the vertex set U, corresponding to n clusters.
- Find the closest pair of objects such that each object is in a different cluster, and add an edge between them.
- Repeat n-k times until there are exactly k clusters.

Key observation. This procedure is precisely Kruskal's algorithm (except we stop when there are k connected components).

Remark. Equivalent to finding an MST and deleting the k-1 most expensive edges.

Greedy Clustering Algorithm: Analysis

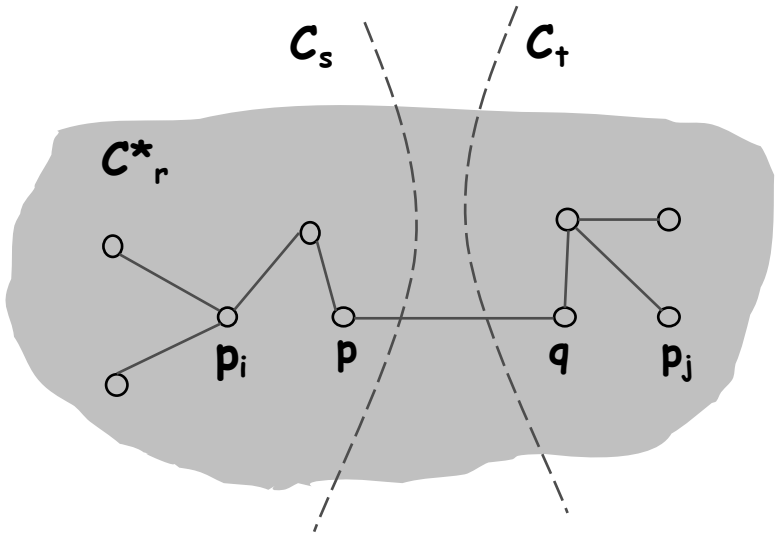
Theorem. Let C^* denote the clustering $C^*_1, \ldots, C^*_k$ formed by deleting the k-1 most expensive edges of a MST. C^* is a k-clustering of max spacing.

Pf. Let C denote some other clustering $C_1, \ldots, C_k$.

- The spacing of C^* is the length d^* of the $(k-1)^{st}$ most expensive edge.
- Let p_i, p_j be in the same cluster in C^*, say C^*_r, but different clusters in C, say C_s and C_t.
- Some edge (p, q) on p_i-p_j path in C^*_r spans two different clusters in C.

- All edges on p_i-p_j path have length $\le d^*$ since Kruskal chose them.

- Spacing of C is $\le d^*$ since p and q are in different clusters.



DIVIDE AND CONQUER

[KT5, CLRS4]

Divide-and-Conquer

Divide-and-conquer.

- Break up problem into several parts.
- Solve each part recursively.
- Combine solutions to sub-problems into overall solution.

Most common usage.

- Break up problem of size n into two equal parts of size $\frac{1}{2}n$.
- Solve two parts recursively.
- Combine two solutions into overall solution in linear time.

Consequence.

- Brute force: n^2.
- Divide-and-conquer: n log n.

Sorting

Sorting. Given n elements, rearrange in ascending order.

Applications.

- Sort a list of names.
- Display Google PageRank results.
- List RSS news items in reverse chronological order.

obvious applications

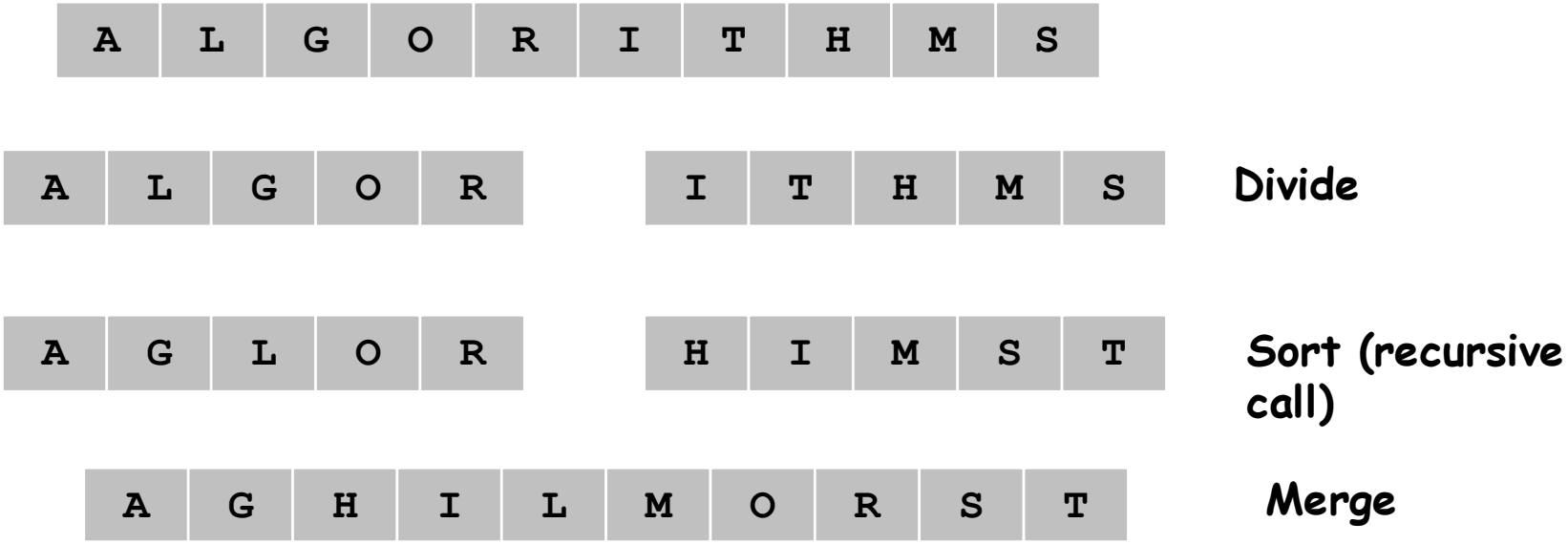
- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

problems become easy once items are in sorted order

Mergesort

Mergesort.

- Divide array into two halves.
- Recursively sort each half (mergesort each half).
- Merge two halves to make sorted whole.

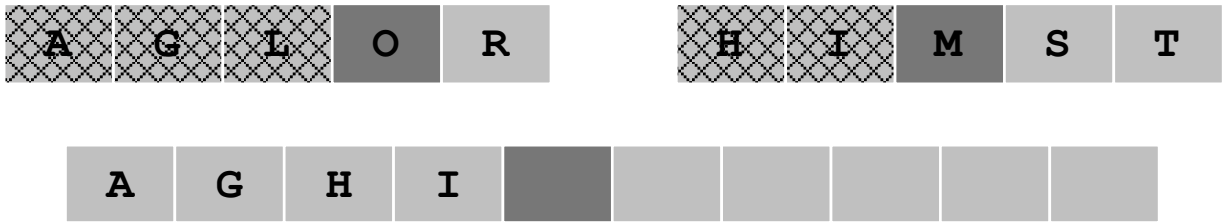


Merging

Merging. Combine two pre-sorted lists into a sorted whole.

How to merge efficiently?

- Linear number of comparisons.
- Use temporary array.



Challenge for the bored. In-place merge. [Kronrud, 1969]

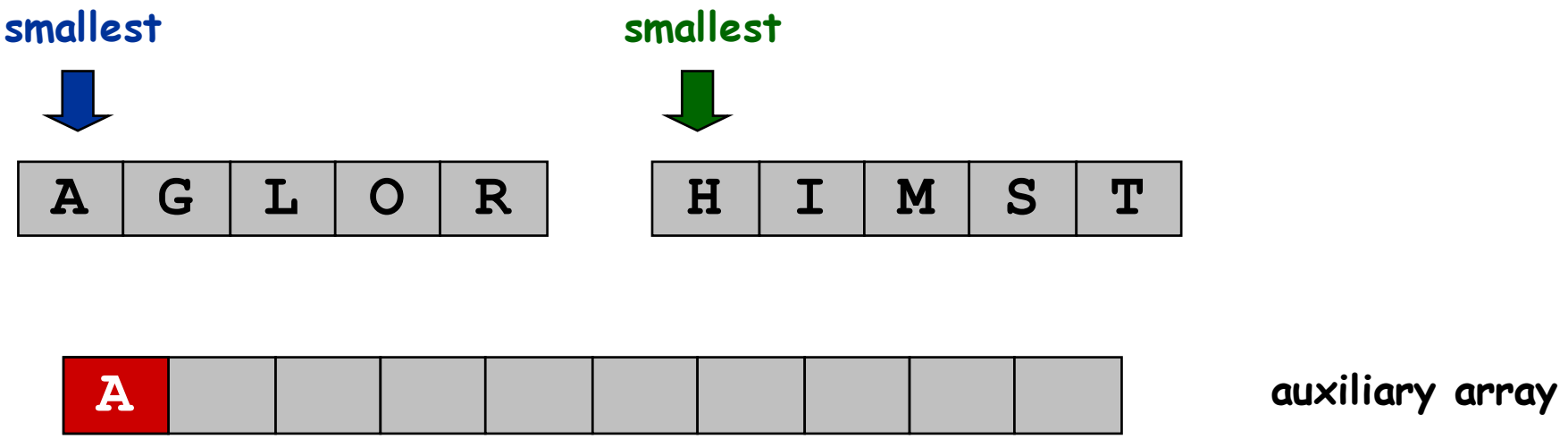
↑

using only a constant amount of extra storage

Merging

Merge: Combine two pre-sorted lists into a sorted whole.

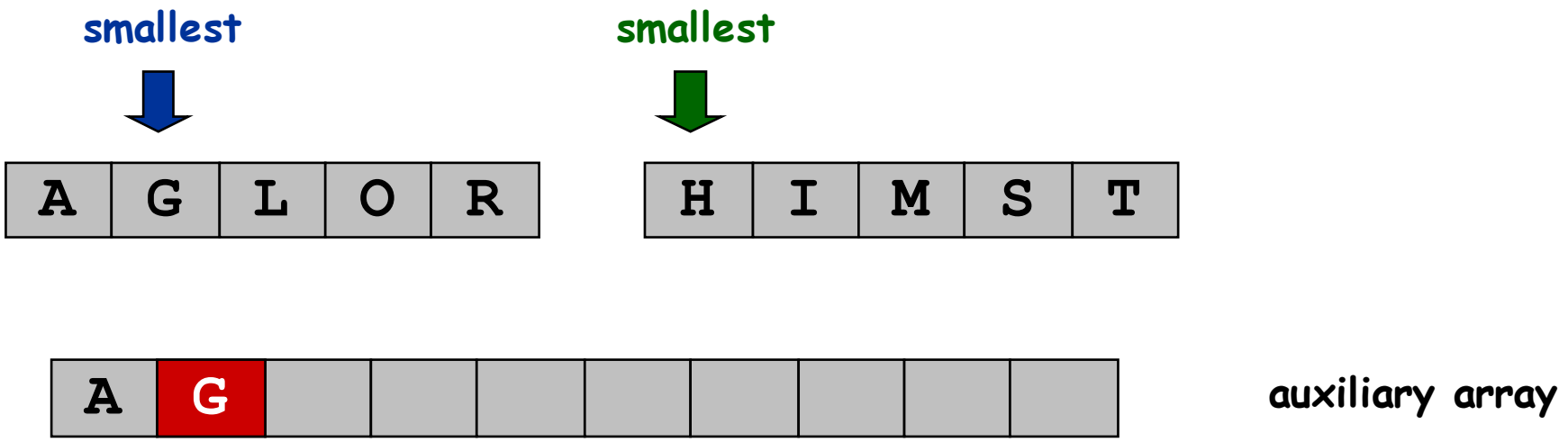
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

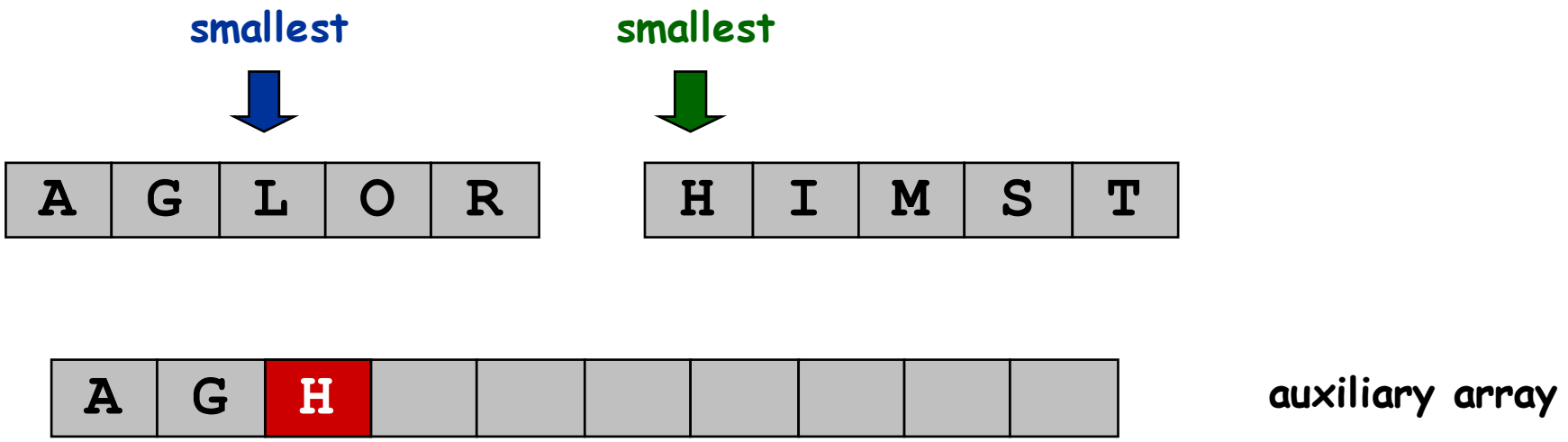
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

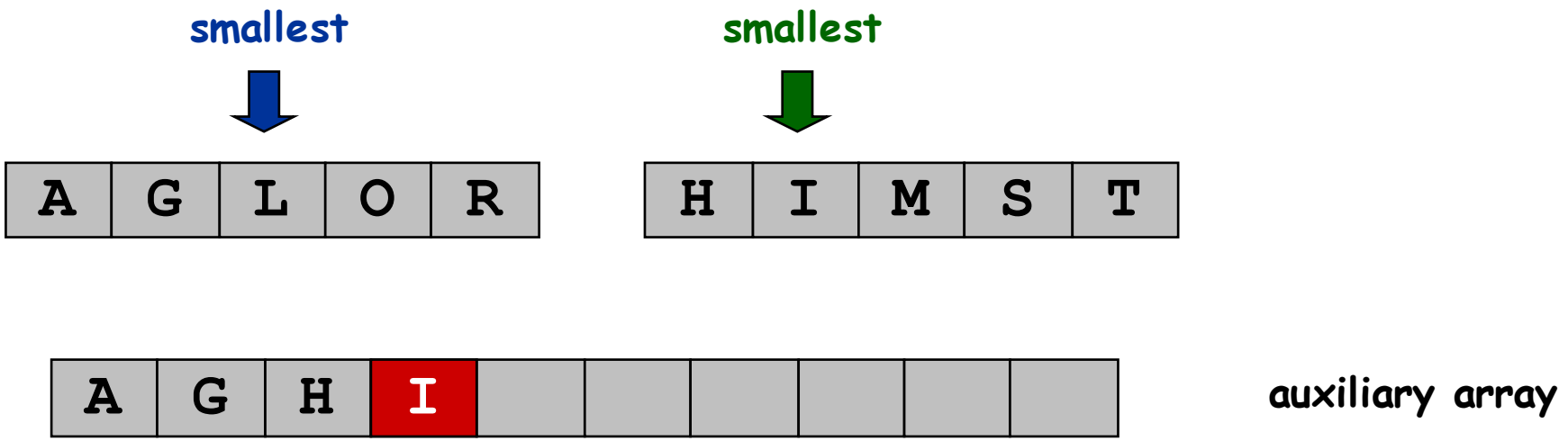
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

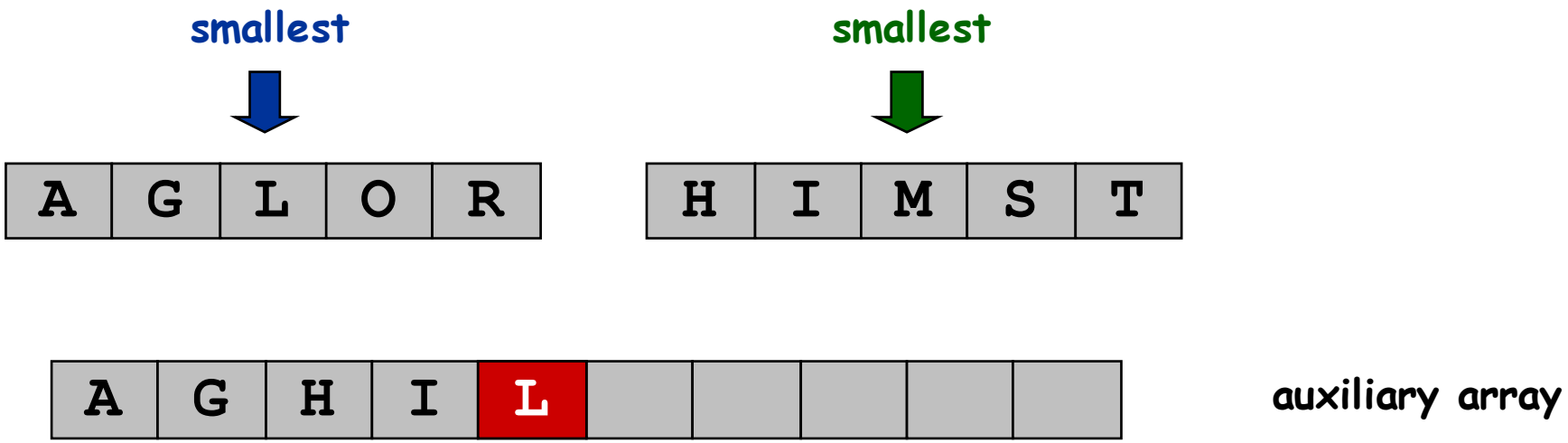
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

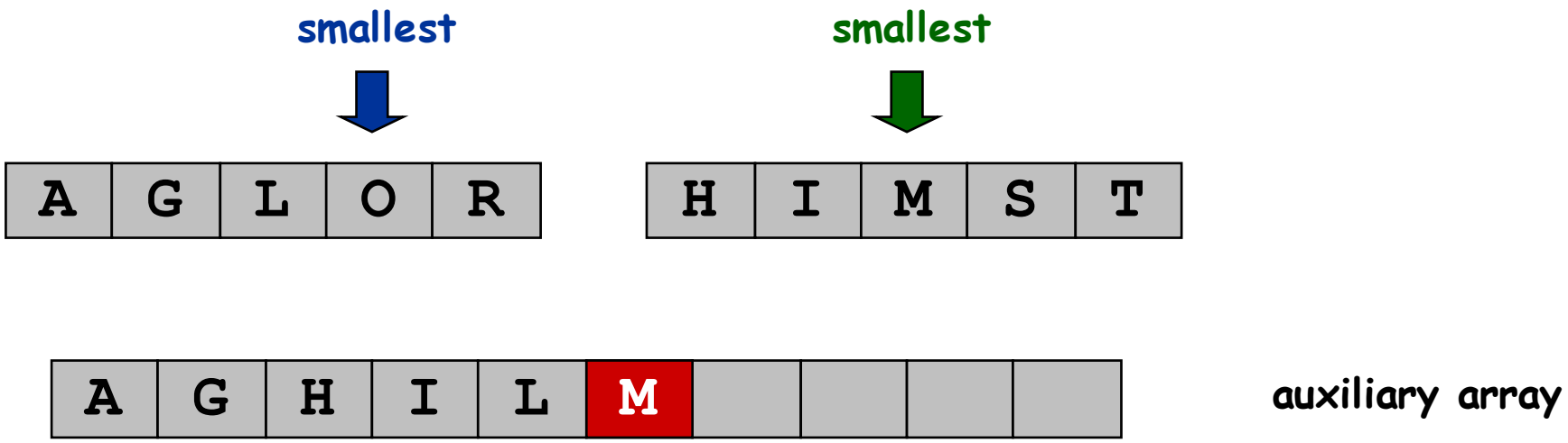
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

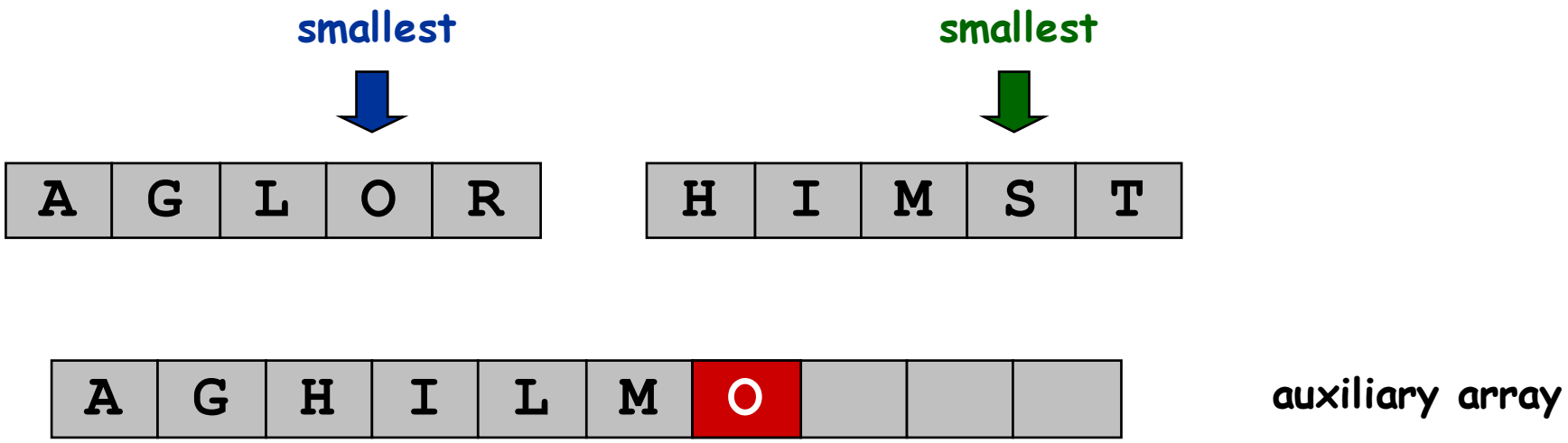
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

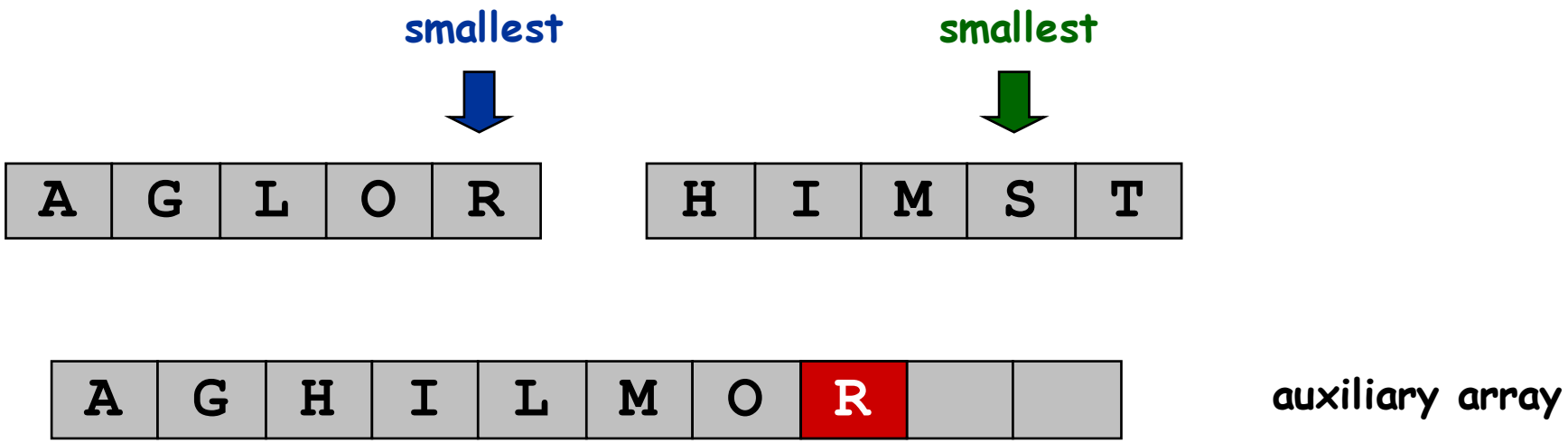
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

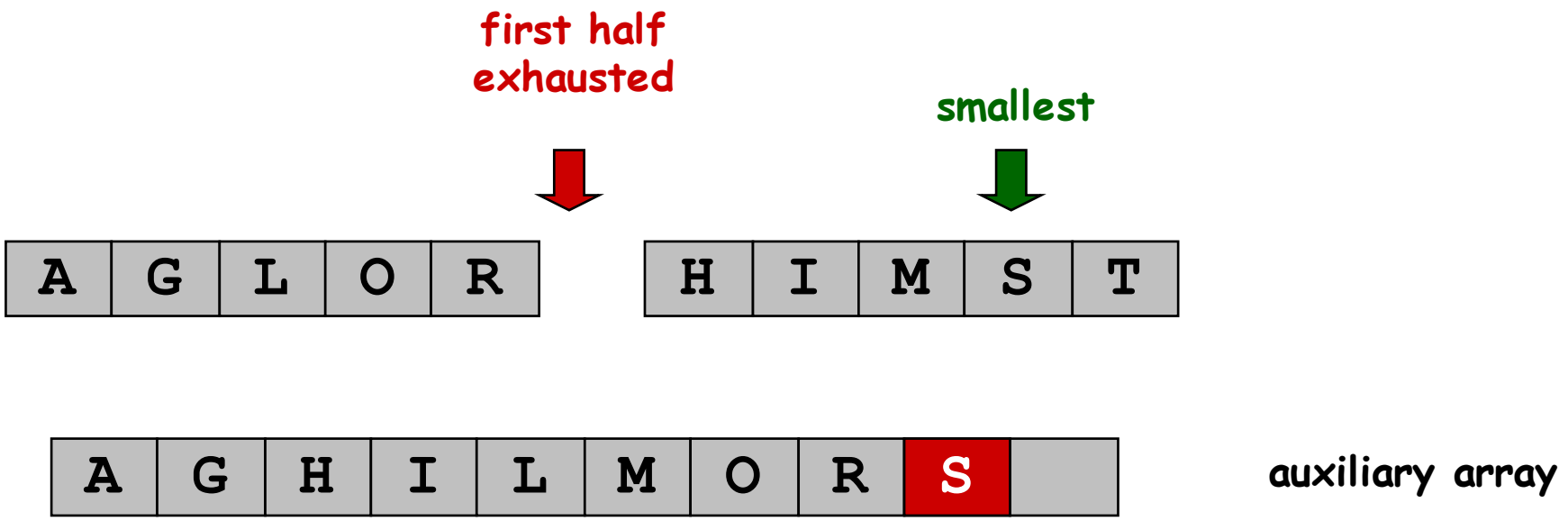
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

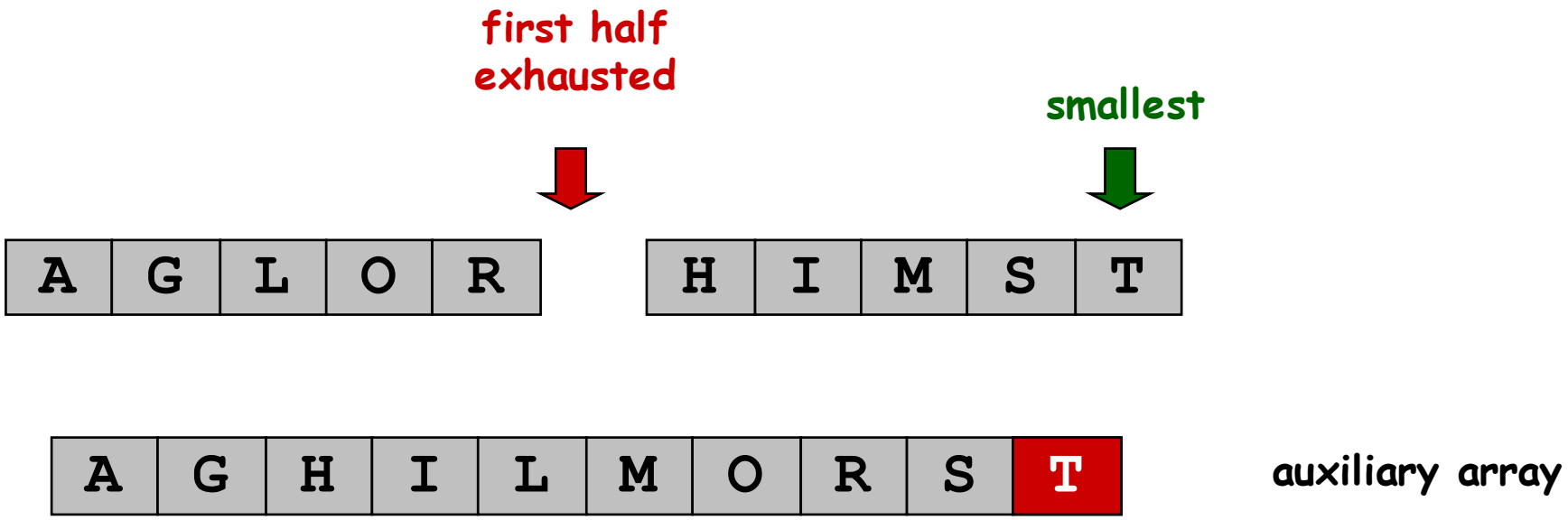
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

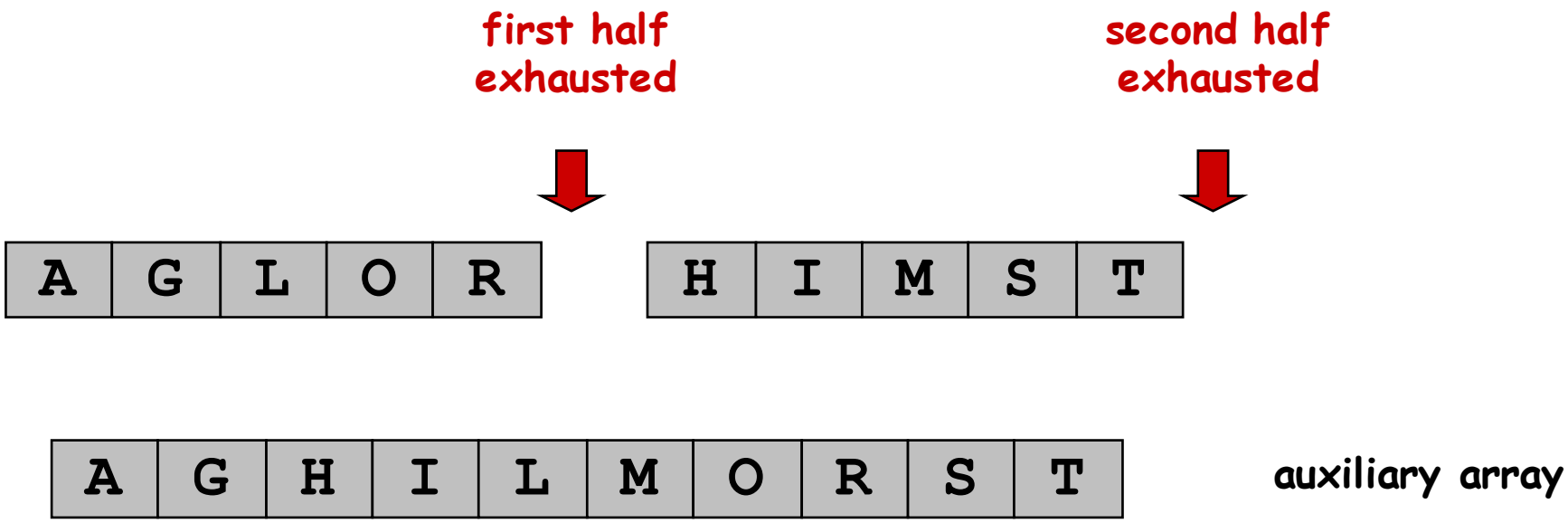
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

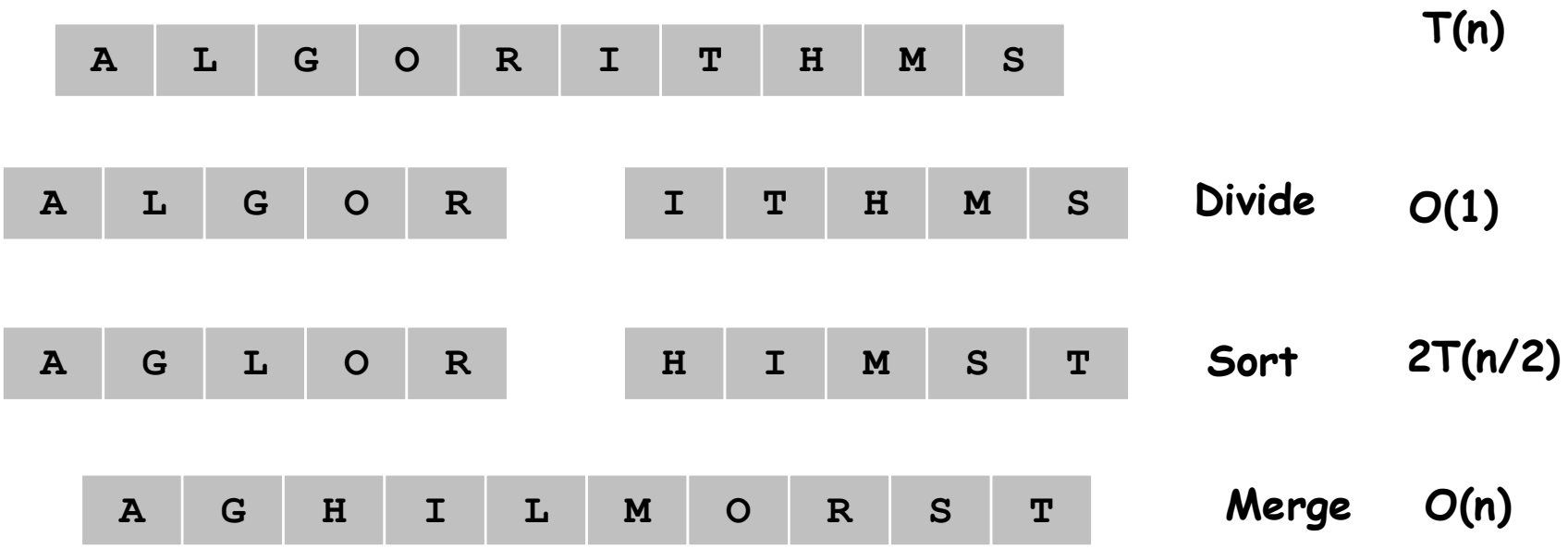
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Mergesort

Mergesort.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.



Recurrence Relation

Def. T(n) = number of comparisons used to mergesort an input of size n.

Mergesort recurrence:

$$\mathrm{T}(n) \le \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Solution. T(n) = $O(n \log_2 n)$.

Guess & Proof by Induction

Assume n is a power of 2 and replace ≤ with =

Claim. If T(n) satisfies this recurrence, then T(n) = n $\log_2$ n.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

↑ assumes n is a power of 2

Pf. (by induction on n)

- Base case: n = 1.
- Inductive hypothesis: T(n) = n $\log_2$ n.
- Goal: show that T(2n) = 2n $\log_2$ (2n).

$$\begin{aligned} T(2n) &= 2T(n) + 2n \\ &= 2n \log_2 n + 2n \\ &= 2n\left(\log_2(2n) - 1\right) + 2n \\ &= 2n \log_2(2n) \end{aligned}$$

Analysis of Mergesort Recurrence

Claim. If T(n) satisfies the following recurrence, then $T(n) \le n \lceil \lg n \rceil$.

$\log_2 n$

$$T(n) \le \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. (by induction on n)

- Base case: n = 1.
- Define $n_1 = \lfloor n / 2 \rfloor$, $n_2 = \lceil n / 2 \rceil$.
- Induction step: assume true for 1, 2, ... , n–1.

$$\begin{aligned} T(n) &\le T(n_1) + T(n_2) + n \\ &\le n_1 \lceil \lg n_1 \rceil + n_2 \lceil \lg n_2 \rceil + n \\ &\le n_1 \lceil \lg n_2 \rceil + n_2 \lceil \lg n_2 \rceil + n \\ &= n \lceil \lg n_2 \rceil + n \\ &\le n(\lceil \lg n \rceil - 1) + n \\ &= n \lceil \lg n \rceil \end{aligned}$$

$$\begin{aligned} n_2 &= \lceil n/2 \rceil \\ &\le \lceil 2^{\lceil \lg n \rceil} / 2 \rceil \\ &= 2^{\lceil \lg n \rceil} / 2 \\ \Rightarrow \lg n_2 &\le \lceil \lg n \rceil - 1 \end{aligned}$$

Master Theorem

Let T(n) be a monotonically increasing function that satisfies

$T(n) = a\,T(n/b) + f(n)$

$T(1) = c$

where $a \geq 1$, $b \geq 2$, $c > 0$.

In divide and conquer algorithms:

a: number of subproblems

n/b: size of each subproblem

f(n): time of combine step

If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Master Theorem: Pitfalls

You cannot use the Master Theorem if

- T(n) is not monotone, e.g. T(n) = sin(x) (not an issue with runtime)
- f(n) is not a polynomial, e.g., $T(n)=2T(n/2)+2^n$
- b cannot be expressed as a constant, e.g.

$$T(n) = aT(\sqrt{n}) + f(n)$$

Master Theorem: Example 1

Let $T(n) = T(n/2) + \frac{1}{2} n^2 + n$. What are the parameters?

a = **1**

b = **2**

d = **2**

Therefore, which condition applies?

$1 < 2^2$, case 1 applies

- **We conclude that**

$$T(n) \in \Theta(n^d) = \Theta(n^2)$$

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Master Theorem: Example 2

Let T(n)= 2 T(n/4) + √n + 42. What are the parameters?

a = **2**

b = **4**

d = **1/2**

Therefore, which condition applies?

$2 = 4^{1/2}$, case 2 applies

- **We conclude that**

$$T(n) \in \Theta(n^d \log n) = \Theta(\log n \sqrt{n})$$

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Master Theorem: Example 3

Let T(n)= 3 T(n/2) + 3/4n + 1. What are the parameters?

a = **3**

b = **2**

d = **1**

Therefore, which condition applies?

$3 > 2^1$, case 3 applies

- **We conclude that**

$$T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3})$$

- **Note that $\log_2 3 \approx 1.584\ldots$, can we say that $T(n) \in \Theta(n^{1.584})$**

No, because $\log_2 3 \approx 1.5849\ldots$ and $n^{1.584} \notin \Theta(n^{1.5849})$

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Matrix Multiplication

Matrix multiplication. Given two n-by-n matrices A and B, compute $C = AB$.

Grade-school. $\Theta(n^3)$ arithmetic operations.

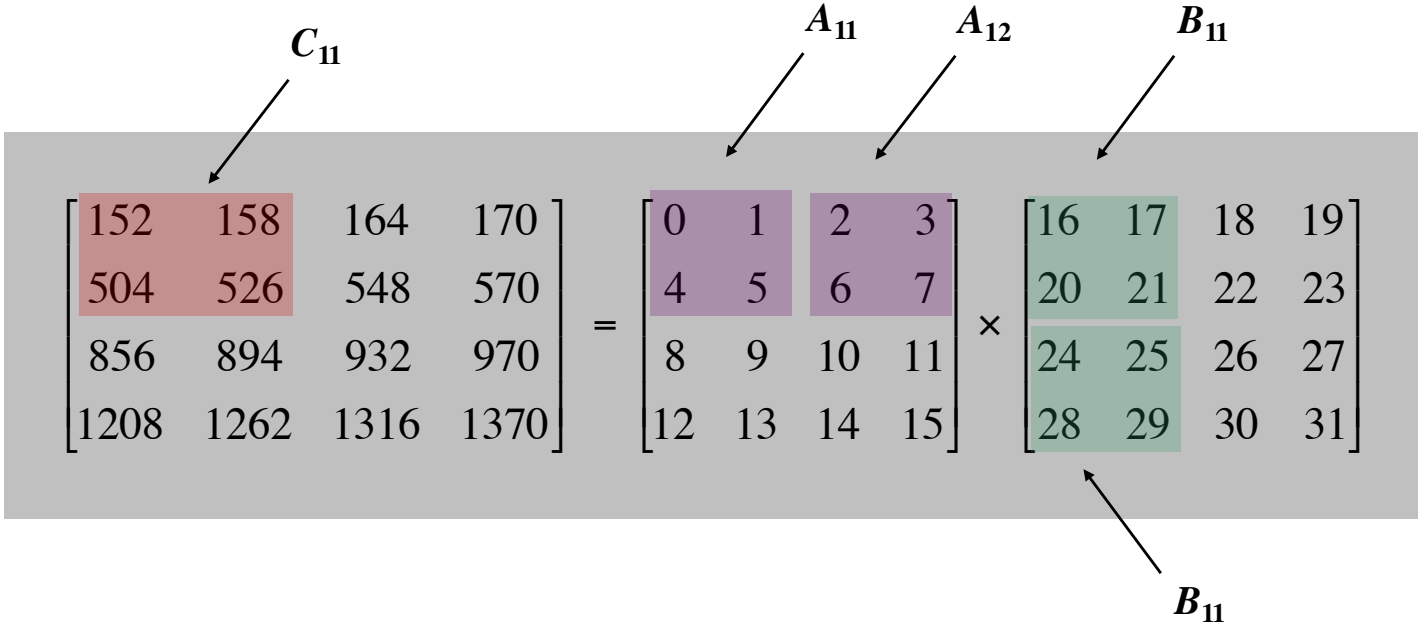
$$c_{ij} = \sum_{k=1}^{n} a_{ik}\, b_{kj}$$

$$\begin{bmatrix} c_{11} & c_{12} & \text{L} & c_{1n} \\ c_{21} & c_{22} & \text{L} & c_{2n} \\ \text{M} & \text{M} & \text{O} & \text{M} \\ c_{n1} & c_{n2} & \text{L} & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \text{L} & a_{1n} \\ a_{21} & a_{22} & \text{L} & a_{2n} \\ \text{M} & \text{M} & \text{O} & \text{M} \\ a_{n1} & a_{n2} & \text{L} & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \text{L} & b_{1n} \\ b_{21} & b_{22} & \text{L} & b_{2n} \\ \text{M} & \text{M} & \text{O} & \text{M} \\ b_{n1} & b_{n2} & \text{L} & b_{nn} \end{bmatrix}$$

$$\begin{bmatrix} .59 & .32 & .41 \\ .31 & .36 & .25 \\ .45 & .31 & .42 \end{bmatrix} = \begin{bmatrix} .70 & .20 & .10 \\ .30 & .60 & .10 \\ .50 & .10 & .40 \end{bmatrix} \times \begin{bmatrix} .80 & .30 & .50 \\ .10 & .40 & .10 \\ .10 & .30 & .40 \end{bmatrix}$$

Q. Is grade-school matrix multiplication algorithm optimal?

Block Matrix Multiplication



$$\begin{bmatrix} 152 & 158 & 164 & 170 \\ 504 & 526 & 548 & 570 \\ 856 & 894 & 932 & 970 \\ 1208 & 1262 & 1316 & 1370 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \times \begin{bmatrix} 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 \\ 28 & 29 & 30 & 31 \end{bmatrix}$$

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} = \begin{bmatrix} 0 & 1 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 16 & 17 \\ 20 & 21 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \times \begin{bmatrix} 24 & 25 \\ 28 & 29 \end{bmatrix} = \begin{bmatrix} 152 & 158 \\ 504 & 526 \end{bmatrix}$$

Matrix Multiplication: Warmup

To multiply two n-by-n matrices A and B:

- Divide: partition A and B into $\frac{1}{2}n$-by-$\frac{1}{2}n$ blocks.
- Conquer: multiply 8 pairs of $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices, recursively.
- Combine: add appropriate products using 4 matrix additions.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

$$T(n) = \underbrace{8T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, form submatrices}} \Rightarrow T(n) = \Theta(n^3)$$

Fast Matrix Multiplication

Key idea. multiply 2-by-2 blocks with only 7 multiplications.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

$$\begin{aligned} P_1 &= A_{11} \times (B_{12} - B_{22}) \\ P_2 &= (A_{11} + A_{12}) \times B_{22} \\ P_3 &= (A_{21} + A_{22}) \times B_{11} \\ P_4 &= A_{22} \times (B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ P_6 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\ P_7 &= (A_{11} - A_{21}) \times (B_{11} + B_{12}) \end{aligned}$$

To multiply two n-by-n matrices A and B: [Strassen 1969]

- Divide: partition A and B into $\frac{1}{2}n$-by-$\frac{1}{2}n$ blocks.
- Compute: 14 $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices via 10 matrix additions.
- Conquer: multiply 7 pairs of $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices, recursively.
- Combine: 7 products into 4 terms using 8 matrix additions.

Fast Matrix Multiplication: Strassen

To multiply two n-by-n matrices A and B: [Strassen 1969]

- Divide: partition A and B into $\frac{1}{2}n$-by-$\frac{1}{2}n$ blocks.
- Compute: 14 $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices via 10 matrix additions.
- Conquer: multiply 7 pairs of $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices, recursively.
- Combine: 7 products into 4 terms using 8 matrix additions.

Analysis.

- Assume n is a power of 2.
- $T(n)$ = # arithmetic operations.

$$T(n) = \underbrace{7T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, subtract}} \Rightarrow T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$$

Common misperception. *"Strassen is only a theoretical curiosity."*

- Apple reports 8x speedup on G4 Velocity Engine when $n \approx 2{,}500$.
- Range of instances where it's useful is a subject of controversy.

Fast Matrix Multiplication: Theory

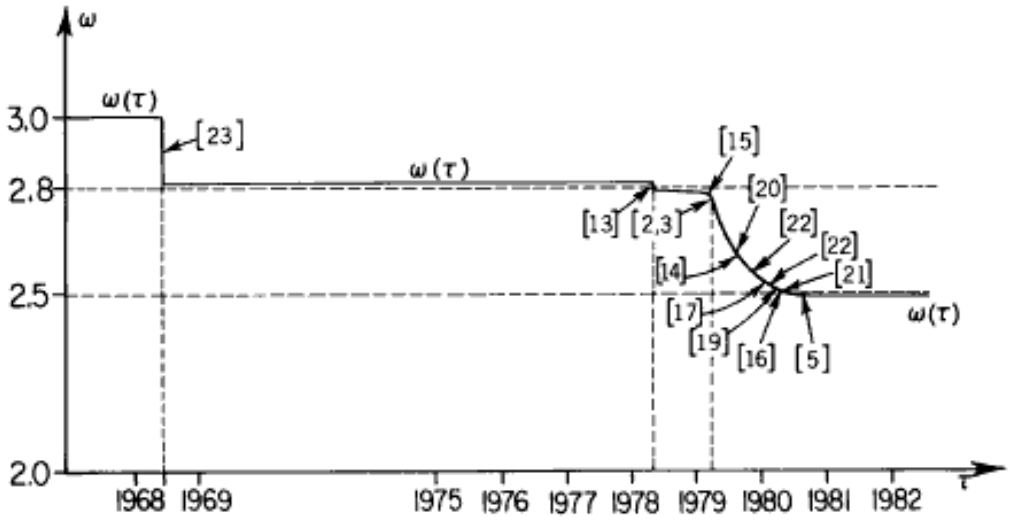


FIG. 1. *ω(t) is the best exponent announced by time τ.*

Best known. $O(n^{2.376})$ [Coppersmith-Winograd, 1987]

Conjecture. $O(n^{2+\varepsilon})$ for any $\varepsilon > 0$.

Caveat. Theoretical improvements to Strassen are progressively less practical.

Dynamic Programming

[KT6, CLRS15, BRV4]

Algorithmic Paradigms

Greedy. Build up a solution incrementally, myopically optimizing some local criterion. (not trying all options but can prove that greedy choice results optimal solution at the end)

Divide-and-conquer. Break up a problem into *non-overlapping* sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

Dynamic programming. Break up a problem into a series of *overlapping* sub-problems, and build up solutions to larger sub-problems from smaller subproblems, (*reusing* solutions of encountered subproblems as much as possible).

Dynamic programming: algorithms which systematically search all possibilities (thus guaranteeing correctness) while storing results to avoid recomputing (thus providing efficiency).

Dynamic Programming Applications

Areas.

- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, compilers, systems, ….

Some famous dynamic programming algorithms.

- Shortest paths - Bellman-Ford
- Comparing two files - Unix diff
- Hidden Markov models - Viterbi
- Genetic sequence alignment - Smith-Waterman
- Parsing context free grammars - Cocke-Kasami-Younger

Dynamic Programming

1) Show problem has optimal substructure: the optimal solution can be constructed from optimal solutions to subproblems (recurrence relation).

2) Show subproblems are overlapping, i.e. subproblems may be encountered many times but the total number of distinct subproblems is polynomial

3) Construct an algorithm that computes the optimal solution to each subproblem only once, and reuses the stored result all other times

4) Show that time and space complexity is polynomial

Coin-changing problem [BRV4.1]

The problem: We want to make change for S cents, and we have infinite supply of each coin in the set Coins={v_1, v_2, …, v_n}, where v_i is the value of the i-th coin. What is the minimum number of coins required to reach value S?

Greedy algorithm:

- sort coins by non-increasing values $v_1 > v_2 > \ldots > v_n$
- R <- S (remaining sum to reach)
- For i=1 to n, { $c_i = \lfloor R/v_i \rfloor$; R <- R – $c_i \times v_i$ }
  (returns c_i coins of value v_i)

Is this optimal?

Optimal for the set {25,10,5,1} but not for {6,4,1} (for instance S=8)

Coin changing problem: DP algorithm

Optimal algorithm. Find z(S,n): reach sum S with coins of value $\{v_1, \ldots, v_n\}$.
Greedy may fail: try to solve more subproblems so that we do not take a bad greedy choice. Must be able to come back to a choice already made and try another set of coins.

Idea: investigate a way to solve a problem in appearance more complex:
find z(T,i), min number of coins to reach T<=S with first i coins;
now we solve S x n problems, but we have a recurrence relation:

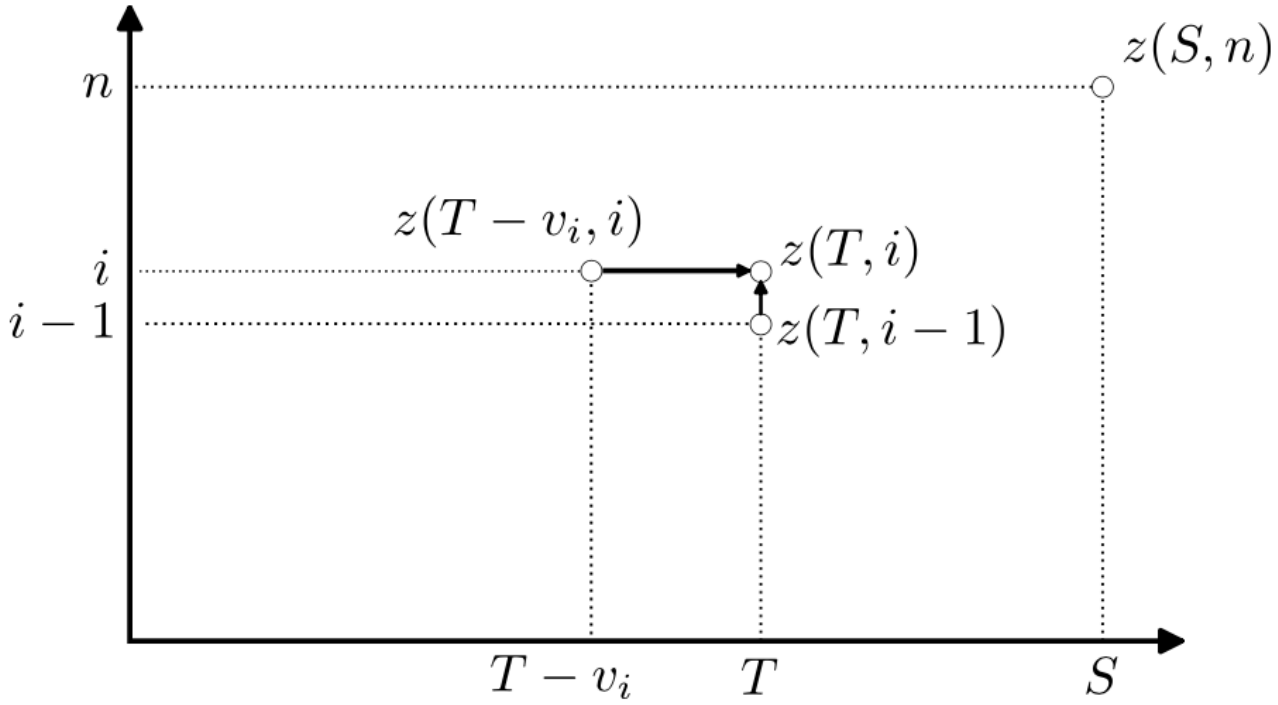
$z(T,i) = \min \{ z(T, i-1)$ (i-th coin not used),
$z(T-v_i, i) + 1$ (i-th coin used at least once) }

Need to initialize the recurrence properly:
$z(T,0) = +\infty$ if $T > 0$ (no more coins)
$z(0,i) = 0$ (we are done)
$z(T,i) = +\infty$ if $T < 0$ (too much change given)

Coin changing problem: implementation

Recursive algorithm: exponential number of computations!

We make « memo » of values already computed, hence using memoization, or use an iterative algorithm so that we always have the values required to compute z(T,i). Check precedence constraints!



Coin changing problem: the algorithm

```
1 for T = 1 to S do
2     z(T, 0) ← +∞      { Initialization: case i = 0 }
3 for i = 0 to n do
4     z(0, i) ← 0       { Initialization: case T = 0 }
5 for i = 1 to n do
6     for T = 1 to S do
7         z(T, i) ← z(T, i − 1)
              { z(T, i − 1) computed at previous iteration, or case i = 0 }
8         if T − v_i ⩾ 0 then
9             z(T, i) ← min(z(T, i), z(T − v_i, i))
                  { z(T − v_i, i) computed earlier in this loop, or case T = 0 }
```

Complexity of DP algorithm: O (n x S)
Greedy algorithm: O (n *log* n)