Final Review

- *greedy algorithms*
- *divide and conquer*
- *dynamic programming*

Final Review

- *greedy algorithms*
- *divide and conquer*
- *dynamic programming*

Greedy algorithms

Problem. Given a digraph $G = (V, E)$, edge lengths $\ell_e \geq 0$, source $s \in V$, and destination $t \in V$, find a shortest directed path from s to t.

Problem. Given a digraph $G = (V, E)$, edge lengths $\ell_e \geq 0$, source $s \in V$, find a shortest directed path from s to every node.

Greedy approach. Maintain a set of explored nodes S for which algorithm has determined $d[u] =$ length of a shortest $s \rightsquigarrow u$ path.

```
DIJKSTRA (V, E, ℓ, s)

FOREACH v ≠ s : π[v] ← ∞, pred[v] ← null; π[s] ← 0.
Create an empty priority queue pq.
FOREACH v ∈ V : INSERT(pq, v, π[v]).
WHILE (IS-NOT-EMPTY(pq))
   u ← DEL-MIN(pq).
   FOREACH edge e = (u, v) ∈ E leaving u:
      IF (π[v] > π[u] + ℓe)
         DECREASE-KEY(pq, v, π[u] + ℓe).
         π[v] ← π[u] + ℓe ; pred[v] ← e.
```

Greedy algorithms

Performance. Depends on PQ: n INSERT, n DELETE-MIN, $\leq m$ DECREASE-KEY.

- Array implementation optimal for dense graphs. ⟵ $\Theta(n^2)$ edges
- Binary heap much faster for sparse graphs. ⟵ $\Theta(n)$ edges
- 4-way heap worth the trouble in performance-critical situations.

priority queue	INSERT	DELETE-MIN	DECREASE-KEY	total
node-indexed array (A[i] = priority of i)	$O(1)$	$O(n)$	$O(1)$	$O(n^2)$
binary heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(m \log n)$
d-way heap (Johnson 1975)	$O(d \log_d n)$	$O(d \log_d n)$	$O(\log_d n)$	$O(m \log_{m/n} n)$
Fibonacci heap (Fredman-Tarjan 1984)	$O(1)$	$O(\log n)$ †	$O(1)$ †	$O(m + n \log n)$
integer priority queue (Thorup 2004)	$O(1)$	$O(\log \log n)$	$O(1)$	$O(m + n \log \log n)$

Final Review

- *greedy algorithms*
- *divide and conquer*
- *dynamic programming*

Divide and conquer

Divide-and-conquer.

so that you can use recursive call to solve it.

Three main steps

- Divide up problem into several subproblems (of the same kind).
- Solve (conquer) each subproblem recursively.
- Combine solutions to subproblems into overall solution.

Most common usage.

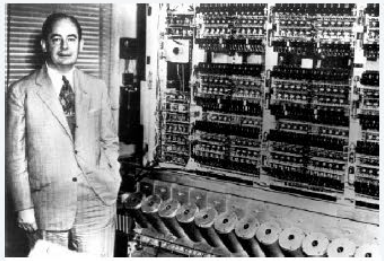
- Divide problem of size n into two subproblems of size $n / 2$. ← $O(n)$ time
- Solve (conquer) two subproblems recursively.
- Combine two solutions into overall solution. ← $O(n)$ time

Consequence.

- Brute force: $\Theta(n^2)$.
- Divide-and-conquer: $O(n \log n)$.

Divide and conquer: merge sort

- Recursively sort left half. → call the same function
- Recursively sort right half. → call the same function
- Merge two halves to make sorted whole.



First Draft of a Report on the EDVAC

John von Neumann

input

A	L	G	O	R	I	T	H	M	S

sort left half

A	G	L	O	R	I	T	H	M	S

sort right half

A	G	L	O	R	H	I	M	S	T

merge results

A	G	H	I	L	M	O	R	S	T

Divide and conquer: merge sort

Input. List L of n elements from a totally ordered universe.

Output. The n elements in ascending order.

MERGE-SORT(L)

IF (list L has one element)

    RETURN L.

Divide the list into two halves A and B.

$A \leftarrow$ MERGE-SORT(A). ⟵ $T(n / 2)$

$B \leftarrow$ MERGE-SORT(B). ⟵ $T(n / 2)$

$L \leftarrow$ MERGE(A, B). ⟵ $\Theta(n)$

RETURN L.

next: How should we find the complexity?

Divide and conquer: counting inversions

Music site tries to match your song preferences with others.

- You rank n songs.
- Music site consults database to find people with similar tastes.

Similarity metric: number of inversions between two rankings.

- My rank: $1, 2, \ldots, n$.
- Your rank: $a_1, a_2, \ldots, a_n$.
- Songs i and j are inverted if $i < j$, but $a_i > a_j$.

	A	B	C	D	E
me	1	2	3	4	5
you	1	3	4	2	5

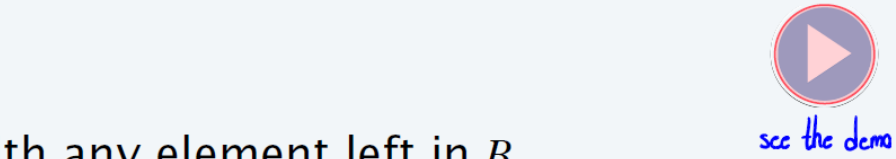
2 inversions: 3-2, 4-2

Brute force: check all $\Theta(n^2)$ pairs.

Divide and conquer: counting inversions

Count inversions (a, b) with $a \in A$ and $b \in B$, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
- If $a_i < b_j$, then a_i is not inverted with any element left in B.
- If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.



Notice that since arrays are sorted. $a_i < 18$ and $b_j < 20 < 23$

count inversions (a, b) with a ∈ A and b ∈ B

merge to form sorted list C

Divide and conquer: counting inversions

Input. List L.

Output. Number of inversions in L and L in sorted order.

SORT-AND-COUNT(L)

IF (list L has one element)

  RETURN $(0, L)$.

Divide the list into two halves A and B.

$(r_A, A) \leftarrow$ SORT-AND-COUNT(A). ⟵ $T(n / 2)$

$(r_B, B) \leftarrow$ SORT-AND-COUNT(B). ⟵ $T(n / 2)$

$(r_{AB}, L) \leftarrow$ MERGE-AND-COUNT(A, B). ⟵ $\Theta(n)$

RETURN $(r_A + r_B + r_{AB}, L)$.

Divide and conquer: randomized quick sort

- Pick a random pivot element $p \in A$. uniformly at random.
- 3-way partition the array into L, M, and R.
- Recursively sort both L and R.

the array A	7	6	12	3	11	8	9	1	4	10	2
		p									
partition A	3	1	4	2	6	7	12	11	8	9	10
sort L	1	2	3	4	6	7	12	11	8	9	10
sort R	1	2	3	4	6	7	8	9	10	11	12
the sorted array A	1	2	3	4	6	7	8	9	10	11	12

Divide and conquer: randomized quick sort

- Pick a random pivot element $p \in A$.
- 3-way partition the array into L, M, and R.
- Recursively sort both L and R.

RANDOMIZED-QUICKSORT(A)

IF (array A has zero or one element)

  RETURN.

Pick pivot $p \in A$ uniformly at random.

$(L, M, R) \leftarrow$ PARTITION-3-WAY(A, p). ← $\Theta(n)$

RANDOMIZED-QUICKSORT(L). ← $T(i)$

RANDOMIZED-QUICKSORT(R). ← $T(n - i - 1)$

new analysis required
(i is a random variable—depends on p)

Note: In here, there is no merge operation, i.e., we first partition & process, then we conquer.

Divide and conquer: median and selection problems

Selection. Given n elements from a totally ordered universe, find k^{th} smallest.

- Minimum: $k = 1$; maximum: $k = n$.
- Median: $k = \lfloor (n + 1) / 2 \rfloor$.
- $O(n)$ compares for min or max.
- $O(n \log n)$ compares by sorting.
- $O(n \log k)$ compares with a binary heap. ← max heap with k smallest

The height needs not to be larger than log k in heap.

Applications. Order statistics; find the "top k"; bottleneck paths, ...

Q. Can we do it with $O(n)$ compares? → should not be a suprise as is the case for min and max.

A. Yes! Selection is easier than sorting.

Divide and conquer: median and selection problems

- Pick a random pivot element $p \in A$.
- 3-way partition the array into L, M, and R.
- Recur in one subarray—the one containing the k^{th} smallest element.

see the demo!

The whole point is that the subarray that has the k^{th} element is the only subarray worth looking at.

QUICK-SELECT(A, k)

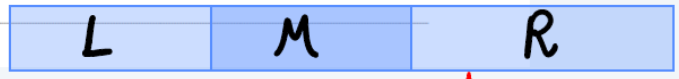
Pick pivot $p \in A$ uniformly at random.

$(L, M, R) \leftarrow$ PARTITION-3-WAY(A, p). ← $\Theta(n)$

IF $(k \le |L|)$ RETURN QUICK-SELECT(L, k). ← $T(i)$

ELSE IF $(k > |L| + |M|)$ RETURN QUICK-SELECT$(R, k - |L| - |M|)$ ← $T(n - i - 1)$

ELSE IF $(k = |L|)$ RETURN p.



$(k-|L|-|M|)^{th}$ element in R = k^{th} element of array (assuming $k > |L|+|M|$)

Divide and conquer: median and selection problems

Goal. Find pivot element p that divides list of n elements into two pieces so that each piece is guaranteed to have $\leq 7/10\ n$ elements.

This is one subproblem $\Rightarrow T(\frac{7}{10}n)$

Why $\frac{7}{10}n$? It just happen to be the case. The main goal is $O(n)$ on both sides.

Q. How to find approximate median in linear time?

A. Recursively compute median of sample of $\leq 2/10\ n$ elements.

This is one subproblem $\Rightarrow T(2/10\ n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(7/10\ n) + T(2/10\ n) + \Theta(n) & \text{otherwise} \end{cases}$$

two subproblems of different sizes!

$$\Rightarrow T(n) = \Theta(n)$$

we'll need to show this

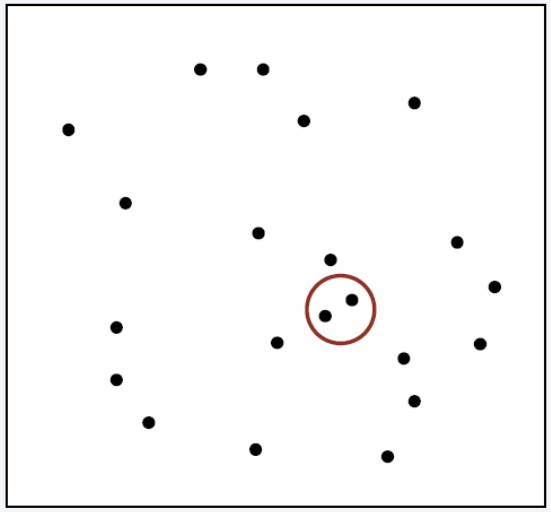
Divide and conquer: closest pair

Closest pair problem. Given n points in the plane, find a pair of points with the smallest Euclidean distance between them.

Fundamental geometric primitive.

- Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.
- Special case of nearest neighbor, Euclidean MST, Voronoi.

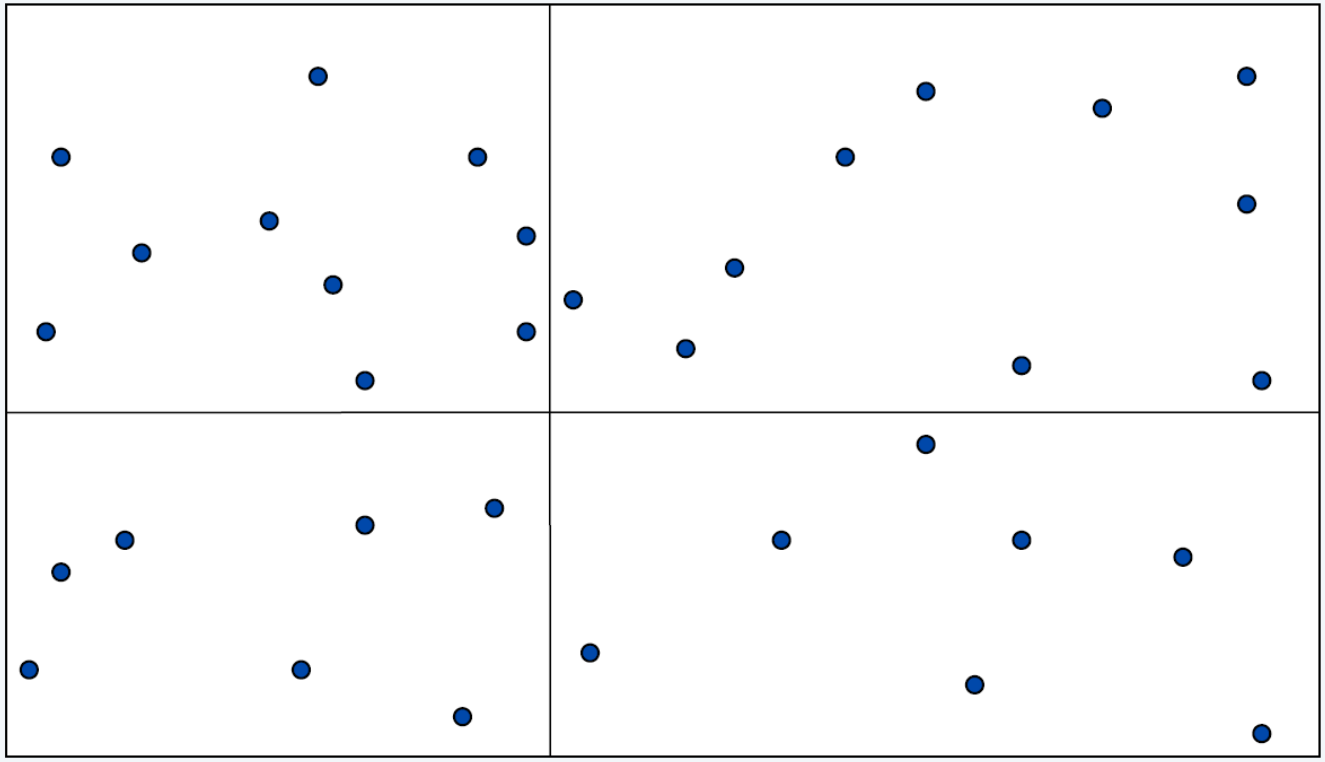
fast closest pair inspired fast algorithms for these problems



Divide and conquer: closest pair

Divide. Subdivide region into 4 quadrants.

Recall: In divide and conquer paradigm, it is important to ensure that subproblems are balanced

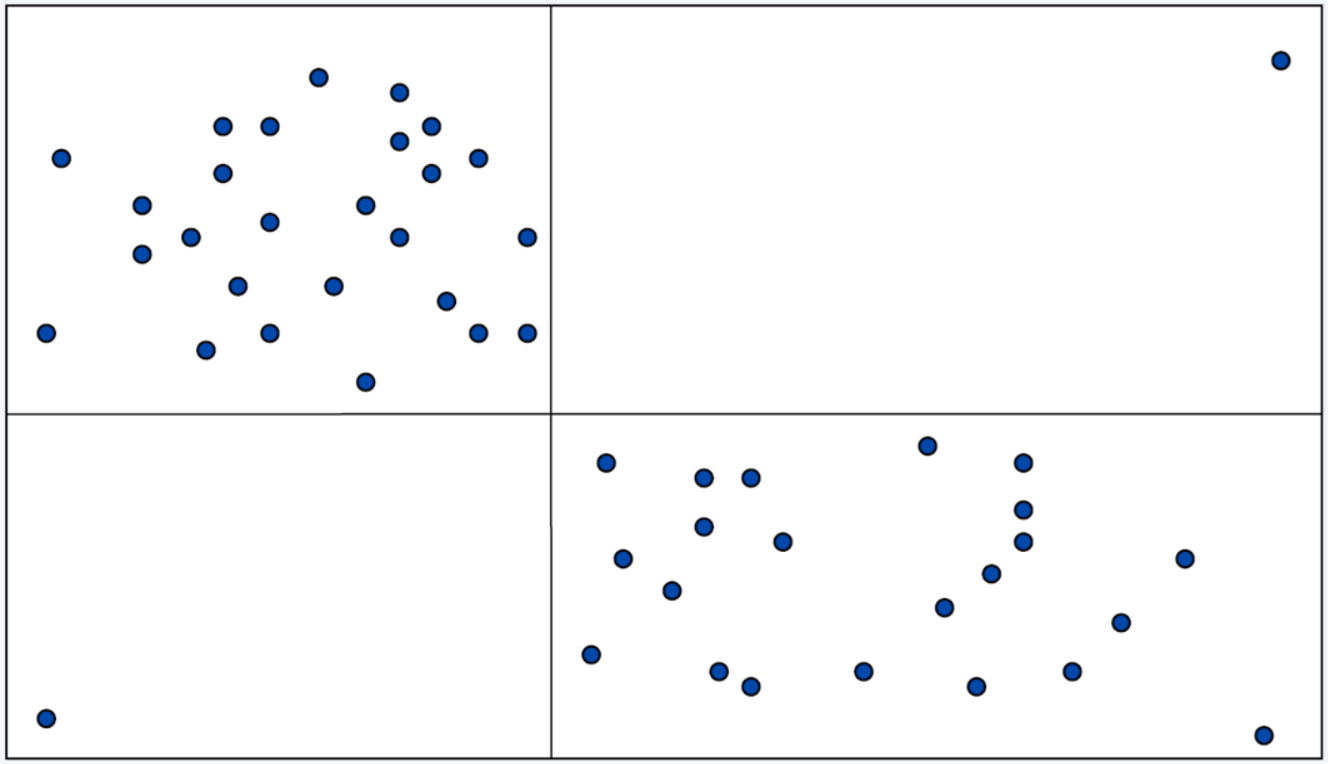


Divide and conquer: closest pair

Divide. Subdivide region into 4 quadrants.

Obstacle. Impossible to ensure $n/4$ points in each piece.

Not balanced

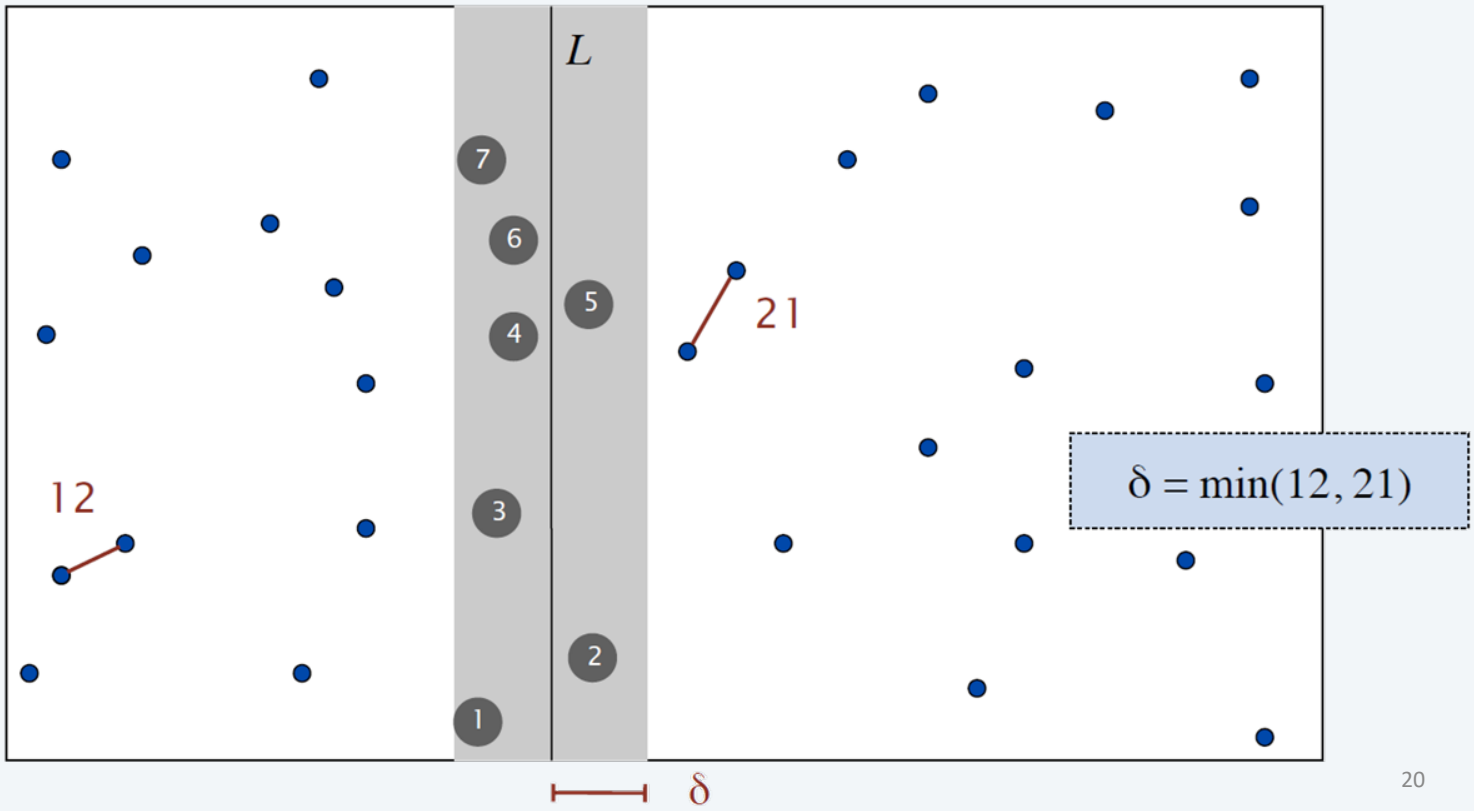


Divide and conquer: closest pair

Find closest pair with one point in each side, assuming that distance $< \delta$.

- Observation: suffices to consider only those points within δ of line L.
- Sort points in 2δ-strip by their y-coordinate.
- Check distances of only those points within 7 positions in sorted list!

why?



Divide and conquer: closest pair

CLOSEST-PAIR($p_1, p_2, \ldots, p_n$)

Compute vertical line L such that half the points are on each side of the line. ← $O(n)$

$\delta_1 \leftarrow$ CLOSEST-PAIR(points in left half). ← $T(n / 2)$

$\delta_2 \leftarrow$ CLOSEST-PAIR(points in right half). ← $T(n / 2)$

$\delta \leftarrow \min \{ \delta_1 , \delta_2 \}$.

Delete all points further than δ from line L. ← $O(n)$

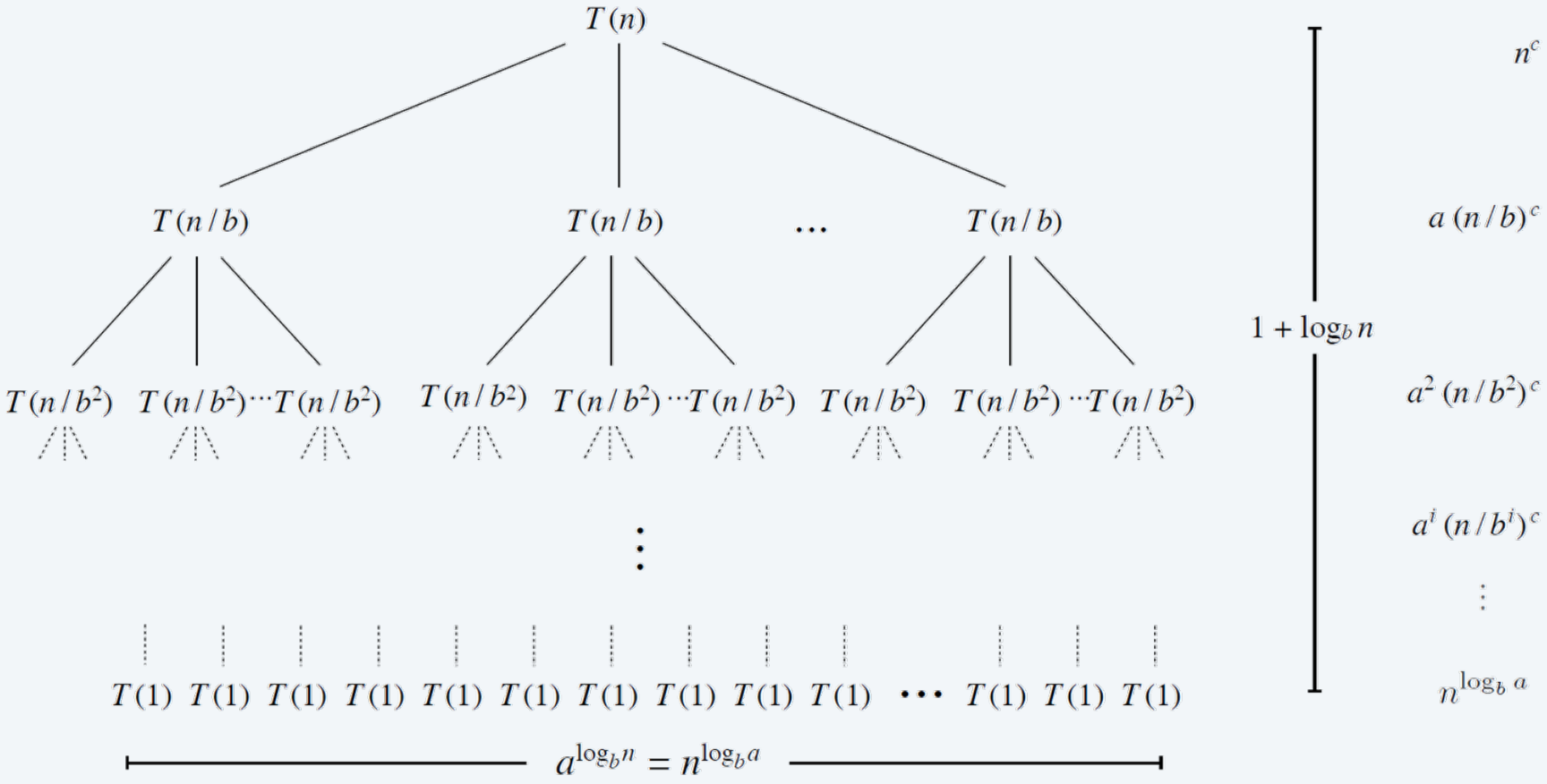
Sort remaining points by y-coordinate. ← $O(n \log n)$

Scan points in y-order and compare distance between each point and next 7 neighbors. If any of these distances is less than δ, update δ. ← $O(n)$

RETURN δ.

Divide and conquer: recursion tree

Suppose $T(n)$ satisfies $T(n) = a\,T(n/b) + n^c$ with $T(1) = 1$, for n a power of b.



$$r = a / b^c \qquad T(n) = n^c \sum_{i=0}^{\log_b n} r^i$$

Divide and conquer: recursion tree

Suppose $T(n)$ satisfies $T(n) = a\,T(n/b) + n^c$ with $T(1) = 1$, for n a power of b.

Let $r = a / b^c$. Note that $r < 1$ iff $c > \log_b a$.

$$T(n) = n^c \sum_{i=0}^{\log_b n} r^i = \begin{cases} \Theta(n^c) & \text{if } r < 1 \quad c > \log_b a \\ \Theta(n^c \log n) & \text{if } r = 1 \quad c = \log_b a \\ \Theta(n^{\log_b a}) & \text{if } r > 1 \quad c < \log_b a \end{cases}$$

cost dominated by cost of root

cost evenly distributed in tree

cost dominated by cost of leaves

Geometric series.

- If $0 < r < 1$, then $1 + r + r^2 + r^3 + \ldots + r^k \le 1 / (1 - r)$.
- If $r = 1$, then $1 + r + r^2 + r^3 + \ldots + r^k = k + 1$.
- If $r > 1$, then $1 + r + r^2 + r^3 + \ldots + r^k = (r^{k+1} - 1) / (r - 1)$.

Divide and conquer: master theorem

Master theorem. Let $a \geq 1$, $b \geq 2$, and $c \geq 0$ and suppose that $T(n)$ is a function on the non-negative integers that satisfies the recurrence

$$T(n) = a\,T\left(\frac{n}{b}\right) + \Theta(n^c)$$

with $T(0) = 0$ and $T(1) = \Theta(1)$, where n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then,

Case 1. If $c > \log_b a$, then $T(n) = \Theta(n^c)$.
Case 2. If $c = \log_b a$, then $T(n) = \Theta(n^c \log n)$.
Case 3. If $c < \log_b a$, then $T(n) = \Theta(n^{\log_b a})$.


Extensions.

- Can replace Θ with O everywhere.
- Can replace Θ with Ω everywhere.
- Can replace initial conditions with $T(n) = \Theta(1)$ for all $n \leq n_0$ and require recurrence to hold only for all $n > n_0$.

Divide and conquer: integer multiplication

To multiply two n-bit integers x and y:

- Divide x and y into low- and high-order bits.
- To compute middle term $bc + ad$, use identity:

$$bc + ad = ac + bd - (a - b)(c - d)$$

given ac and bd, we only need one recursion call

→ compute bc+ad by recycling other values that are calculated, i.e., ac and bd

Note: To improve the complexity, we need to reduce the number of recursive calls. Notice that

$$x\,y = 2^{2m}\,ac + 2^{m}\,(bc+ad) + bd$$

Hence, we have to calculate ac and bd.

- Multiply only three $\frac{1}{2}n$-bit integers, recursively.

$$m = \lceil n / 2 \rceil$$

$$a = \lfloor x / 2^m \rfloor \quad b = x \bmod 2^m$$

$$c = \lfloor y / 2^m \rfloor \quad d = y \bmod 2^m$$

$$x = \underbrace{1\,0\,0\,0}_{a}\,\underbrace{1\,1\,0\,1}_{b}$$

$$y = \underbrace{1\,1\,1\,0}_{c}\,\underbrace{0\,0\,0\,1}_{d}$$

middle term

$$x\,y = (2^m a + b)(2^m c + d) = 2^{2m}\,ac + 2^m\,(bc + ad) + bd$$

$$= 2^{2m}\,ac + 2^m\,(ac + bd - (a-b)(c-d)) + bd$$

(1) (1) (3) (2) (3)

Divide and conquer: integer multiplication

Integer multiplication. Given two n-bit integers, compute their product.

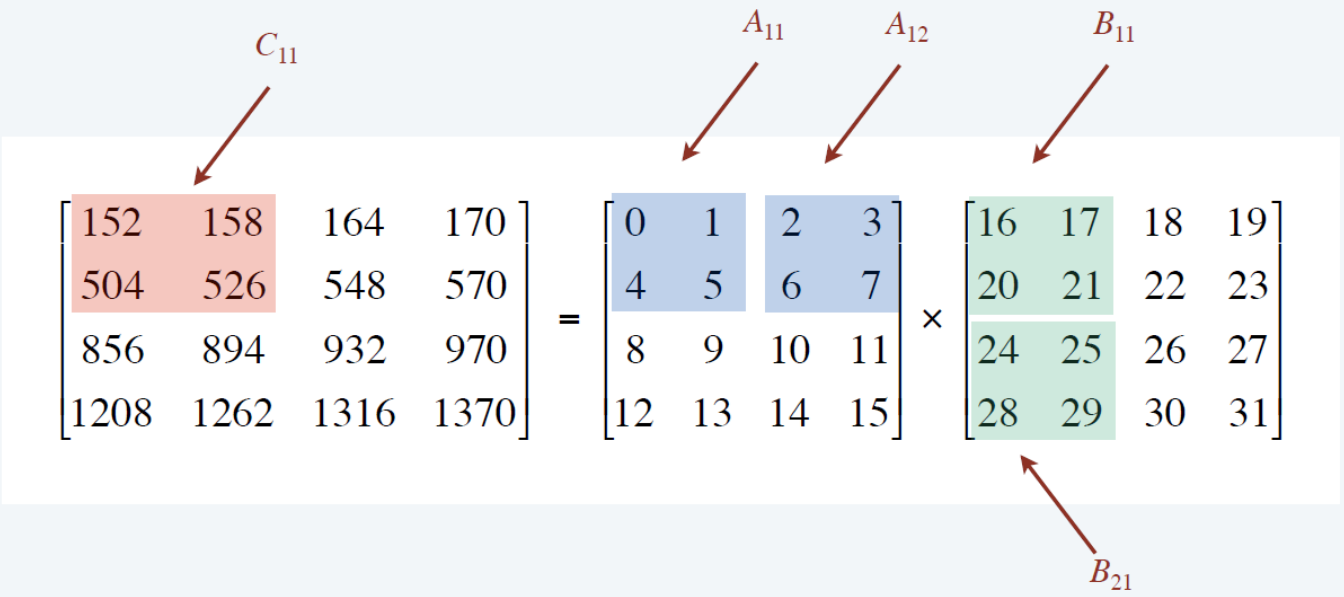
arithmetic problem	formula	bit complexity
integer multiplication	$a \times b$	$M(n)$
integer square	a^2	$\Theta(M(n))$
integer division	$\lfloor a / b \rfloor,\ a \bmod b$	$\Theta(M(n))$
integer square root	$\lfloor \sqrt{a} \rfloor$	$\Theta(M(n))$

$$ab = \frac{(a+b)^2 - a^2 - b^2}{2}$$

integer arithmetic problems with the same bit complexity M(n) as integer multiplication

That is, integer multiplication is fundamental; improving complexity $M(n)$ benefits many arithmetic operations.

Divide and conquer: matrix multiplication



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

Divide and conquer: matrix multiplication

Key idea. Can multiply two 2-by-2 matrices via 7 scalar multiplications (plus 11 additions and 7 subtractions).

scalars

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

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

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

7 scalar multiplications

Pf. $$\begin{aligned} C_{12} &= P_1 + P_2 \\ &= A_{11} \times (B_{12} - B_{22}) + (A_{11} + A_{12}) \times B_{22} \\ &= A_{11} \times B_{12} + A_{12} \times B_{22}. \quad \checkmark \end{aligned}$$

Divide and conquer: matrix multiplication

Key idea. Can multiply two ~~2-by-2~~ *n*-by-*n* matrices via 7 ~~scalar~~ ½*n*-by-½*n* matrix multiplications (plus 11 additions and 7 subtractions).

½*n*-by-½*n* matrices

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

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

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

7 matrix multiplications (of ½*n*-by-½*n* matrices)

Pf.
$$\begin{aligned} C_{12} &= P_1 + P_2 \\ &= A_{11} \times (B_{12} - B_{22}) + (A_{11} + A_{12}) \times B_{22} \\ &= A_{11} \times B_{12} + A_{12} \times B_{22}. \quad \checkmark \end{aligned}$$

Divide and conquer: matrix multiplication

Theorem. Strassen's algorithm requires $O(n^{2.81})$ arithmetic operations to multiply two n-by-n matrices.

Pf.

- When n is a power of 2, apply Case 1 of the master theorem:

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

- When n is not a power of 2, pad matrices with zeros to be n'-by-n', where $n \le n' < 2n$ and n' is a power of 2.

Course Review!

Final Review

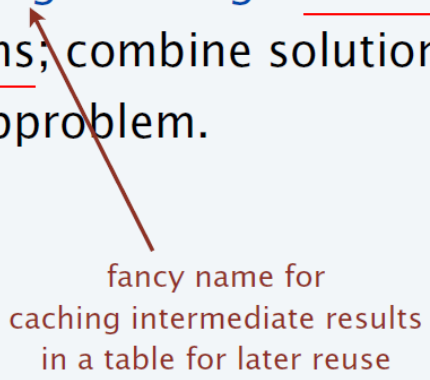
- *greedy algorithms*
- *divide and conquer*
- *dynamic programming*

Dynamic programming

Greed. Process the input in some order, myopically making irrevocable decisions.

Divide-and-conquer. Break up a problem into independent subproblems; solve each subproblem; combine solutions to subproblems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping subproblems; combine solutions to smaller subproblems to form solution to large subproblem.



Dynamic programming: weighted interval scheduling

- Job j starts at s_j, finishes at f_j, and has weight $w_j > 0$.
- Two jobs are compatible if they don't overlap.
- Goal: find max-weight subset of mutually compatible jobs.

Def. $OPT(j)$ = max weight of any subset of mutually compatible jobs for subproblem consisting only of jobs $1, 2, \ldots, j$.

first step: define OPT

Goal. $OPT(n)$ = max weight of any subset of mutually compatible jobs.

Case 1. $OPT(j)$ does not select job j.

- Must be an optimal solution to problem consisting of remaining jobs $1, 2, \ldots, j-1$.

optimal substructure property (proof via exchange argument)

Case 2. $OPT(j)$ selects job j.

- Collect profit w_j.
- Can't use incompatible jobs $\{ p(j)+1, p(j)+2, \ldots, j-1 \}$.
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \ldots, p(j)$.

ecand step: recursive relation

Dynamic programming: weighted interval scheduling

Bellman equation. $$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{ OPT(j-1),\ w_j + OPT(p(j)) \} & \text{if } j > 0 \end{cases}$$

0 → initialization

recursive relation

BRUTE-FORCE $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

Sort jobs by finish time and renumber so that $f_1 \le f_2 \le \ldots \le f_n$. → O(n log n)

Compute $p[1], p[2], \ldots, p[n]$ via binary search. → O(n log n)

RETURN COMPUTE-OPT(n).

COMPUTE-OPT(j)

IF ($j = 0$)

  RETURN 0. → initialization

ELSE

  RETURN max {COMPUTE-OPT($j-1$), w_j + COMPUTE-OPT($p[j]$) }. → recursive relation

Dynamic programming: weighted interval scheduling

Top-down dynamic programming (memoization).

- Cache result of subproblem j in $M[j]$.
- Use $M[j]$ to avoid solving subproblem j more than once.

TOP-DOWN$(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

Sort jobs by finish time and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

Compute $p[1], p[2], \ldots, p[n]$ via binary search.

$M[0] \leftarrow 0$. ← global array

RETURN M-COMPUTE-OPT(n).

→ In top down approach, we use recursive calls to solve the problem.

M-COMPUTE-OPT(j)

IF ($M[j]$ is uninitialized)

$M[j] \leftarrow \max \{$ M-COMPUTE-OPT$(j-1)$, w_j + M-COMPUTE-OPT$(p[j])$ $\}$.

RETURN $M[j]$.

Claim. Memoized version of algorithm takes $O(n \log n)$ time.

Dynamic programming: weighted interval scheduling

Bottom-up dynamic programming. Unwind recursion.

In bottom_up approach we start by solving Bellman equation for smaller values.

BOTTOM-UP$(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p[1], p[2], \ldots, p[n]$.

$M[0] \leftarrow 0$.

previously computed values

FOR $j = 1$ TO n

$M[j] \leftarrow \max \{ M[j-1],\ w_j + M[p[j]] \}$.

use already known values to find $M[j]$.

Note: Bottom_up approach & Top_down approach with memorizatin are implementation details, i.e., have nothing to do with Bellman equation.

Running time. The bottom-up version takes $O(n \log n)$ time.

Dynamic programming: weighted interval scheduling

Q. DP algorithm computes optimal value. How to find optimal solution?

A. Make a second pass by calling FIND-SOLUTION(n).

FIND-SOLUTION(j)

IF $(j = 0)$

  RETURN $\varnothing$.

ELSE IF $(w_j + M[p[j]] > M[j-1])$ → In this case we have picked job j.

  RETURN $\{ j \} \cup$ FIND-SOLUTION($p[j]$).

ELSE

  RETURN FIND-SOLUTION($j-1$).

$$M[j] = \max \{ M[j-1],\ w_j + M[p[j]] \}.$$

→ Given M, we can figure out the optimum solution by checking the Bellman equation.

Analysis. # of recursive calls $\leq n \Rightarrow O(n)$.

Dynamic programming: Kadane's algorithm

Def. $OPT(i)$ = max sum of any subarray of x whose rightmost index is i.

Goal. $\max_i OPT(i)$

Bellman equation.

$$OPT(i) = \begin{cases} x_1 & \text{if } i = 1 \\ \max\{ x_i,\ x_i + OPT(i-1) \} & \text{if } i > 1 \end{cases}$$

take only element i

take element i together with best subarray ending at index $i-1$

Running time. $O(n)$.

Recall that using divide & conquer, the complexity was $O(n \log n)$.

Note: Dynamic programming usually results in better time complexity & worst space complexity compared to divide & conquer.

Dynamic programming: segmented least squares

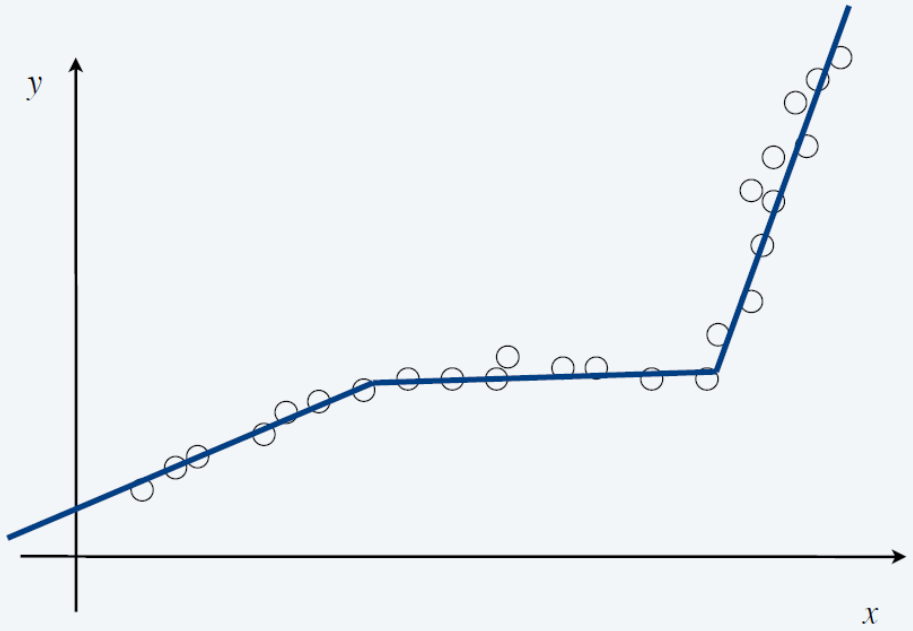
Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes $f(x)$.

this will restrict the # of lines we pick.

Goal. Minimize $f(x) = E + c\,L$ for some constant $c > 0$, where

- E = sum of the sums of the squared errors in each segment.
- L = number of lines.



Dynamic programming: segmented least squares

Notation.

- $OPT(j)$ = minimum cost for points $p_1, p_2, \ldots, p_j$.
- e_{ij} = SSE for for points $p_i, p_{i+1}, \ldots, p_j$. ⇝ that is, we use one line for $p_i, \ldots, p_j$ that minimizes SSE

minimum

To compute $OPT(j)$:

- Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some $i \le j$.
- Cost = $e_{ij} + c + OPT(i-1)$. ⟵ optimal substructure property (proof via exchange argument)

Bellman equation.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{1 \le i \le j} \{ e_{ij} + c + OPT(i-1) \} & \text{if } j > 0 \end{cases}$$

Theorem. [Bellman 1961] DP algorithm solves the segmented least squares problem in $O(n^3)$ time and $O(n^2)$ space.

Remark. Can be improved to $O(n^2)$ time.

Dynamic programming: knapsack problem

Goal. Pack knapsack so as to maximize total value of items taken.

- There are n items: item i provides value $v_i > 0$ and weighs $w_i > 0$.
- Value of a subset of items = sum of values of individual items.
- Knapsack has weight limit of W.

Def. $OPT(i, w)$ = optimal value of knapsack problem with items $1, \ldots, i$, subject to weight limit w.

Goal. $OPT(n, W)$.

Step 1. Define OPT

Bellman equation.

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{\, OPT(i-1, w),\ v_i + OPT(i-1, w - w_i) \,\} & \text{otherwise} \end{cases}$$

initialization

initialization

Theorem. The DP algorithm solves the knapsack problem with n items and maximum weight W in $\Theta(n\,W)$ time and $\Theta(n\,W)$ space.

Dynamic programming: knapsack problem

i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w),\ v_i + OPT(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$

Example: which items would you pick if W = 10? answer in blue. {1, 2, 5}

weight limit w

subset of items 1, …, i	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

OPT(i, w) = optimal value of knapsack problem with items 1, …, i, subject to weight limit w

Dynamic programming: sequence alignment

Goal. Given two strings $x_1 x_2 \ldots x_m$ and $y_1 y_2 \ldots y_n$, find a min-cost alignment.

Def. An alignment M is a set of ordered pairs $x_i - y_j$ such that each character appears in at most one pair and no crossings.

$x_i - y_j$ and $x_{i'} - y_{j'}$ cross if $i < i'$, but $j > j'$

Def. The cost of an alignment M is:

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i : x_i \text{ unmatched}} \delta + \sum_{j : y_j \text{ unmatched}} \delta}_{\text{gap}}$$

Def. $OPT(i, j)$ = min cost of aligning prefix strings $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$.

Goal. $OPT(m, n)$.

Dynamic programming: sequence alignment

Case 1. $OPT(i, j)$ matches $x_i - y_j$.

Pay mismatch for $x_i - y_j$ + min cost of aligning $x_1\, x_2\, \ldots\, x_{i-1}$ and $y_1\, y_2\, \ldots\, y_{j-1}$.

Case 2a. $OPT(i, j)$ leaves x_i unmatched.

Pay gap for x_i + min cost of aligning $x_1\, x_2\, \ldots\, x_{i-1}$ and $y_1\, y_2\, \ldots\, y_j$.

Case 2b. $OPT(i, j)$ leaves y_j unmatched.

Pay gap for y_j + min cost of aligning $x_1\, x_2\, \ldots\, x_i$ and $y_1\, y_2\, \ldots\, y_{j-1}$.

optimal substructure property
(proof via exchange argument)

Bellman equation.

$$OPT(i,j) = \begin{cases} j\delta & \text{if } i = 0 \\ i\delta & \text{if } j = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \end{cases}$$

Dynamic programming: sequence alignment

Theorem. The DP algorithm computes the edit distance (and an optimal alignment) of two strings of lengths m and n in $\Theta(mn)$ time and space.

Pf.

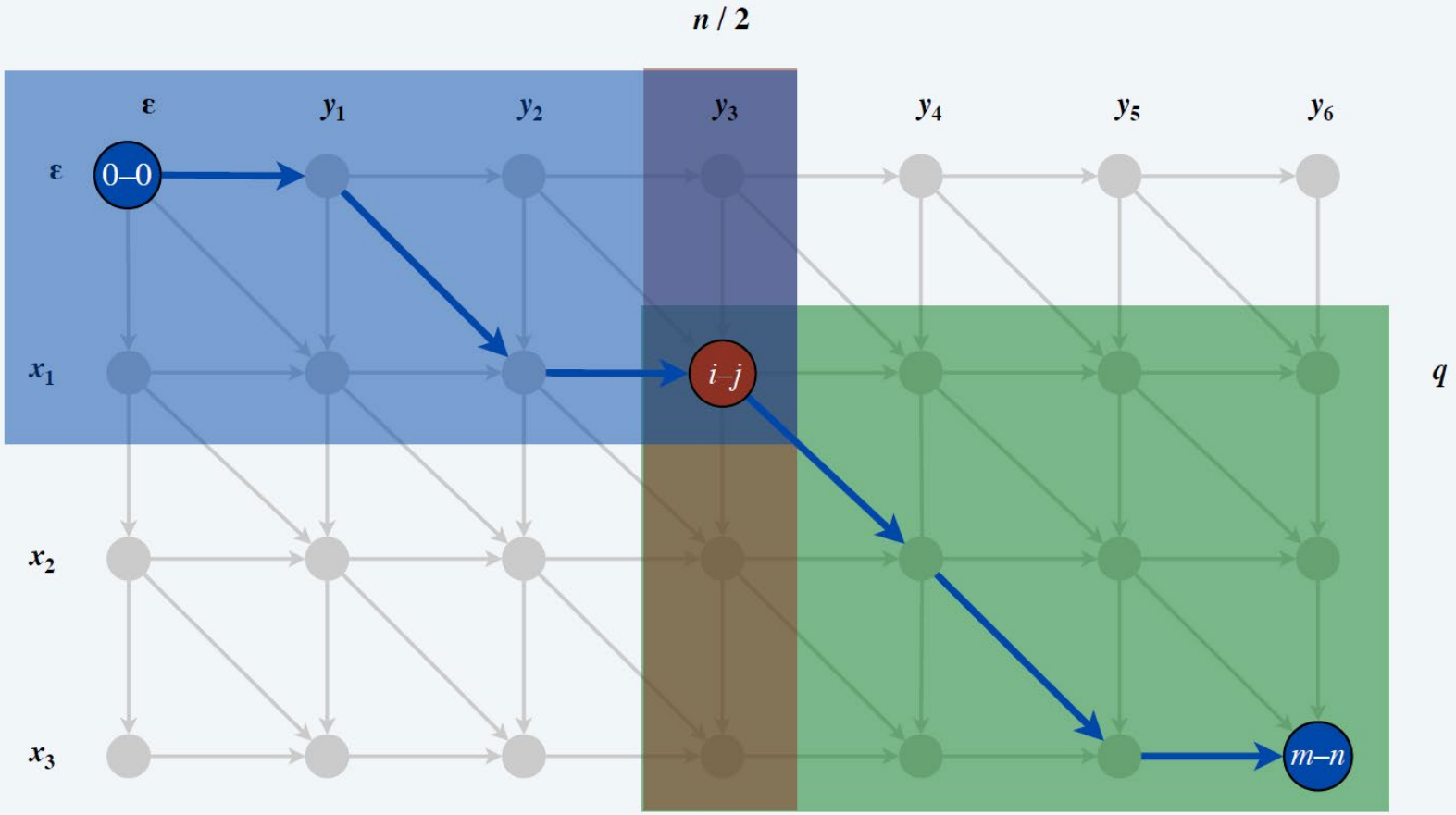
- Algorithm computes edit distance.
- Can trace back to extract optimal alignment itself. ▪

Theorem. [Backurs–Indyk 2015] If can compute edit distance of two strings of length n in $O(n^{2-\varepsilon})$ time for some constant $\varepsilon > 0$, then can solve SAT with n variables and m clauses in $poly(m)\, 2^{(1-\delta)\, n}$ time for some constant $\delta > 0$.

Dynamic programming: sequence alignment

Theorem. [Hirschberg] There exists an algorithm to find an optimal alignment in $O(mn)$ time and $O(m + n)$ space.

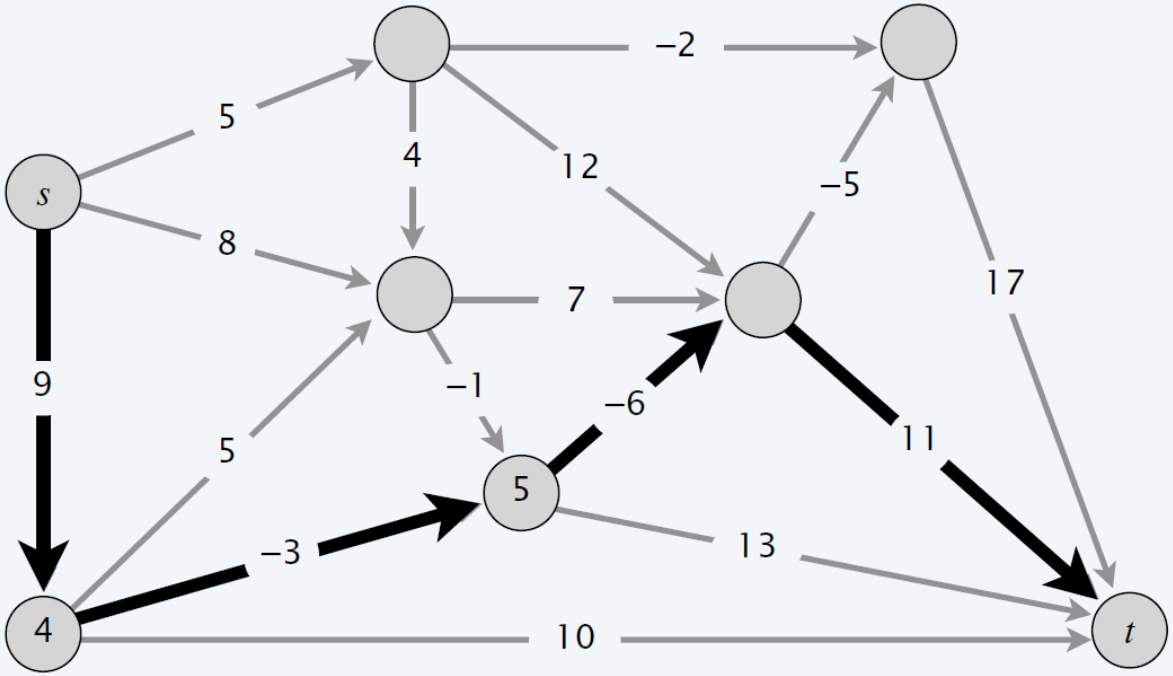
- Clever combination of divide-and-conquer and dynamic programming.
- Inspired by idea of Savitch from complexity theory.



Dynamic programming: shortest path problem

Shortest-path problem. Given a digraph $G = (V, E)$, with arbitrary edge lengths ℓ_{vw}, find shortest path from source node s to destination node t.

assume there exists a path from every node to t

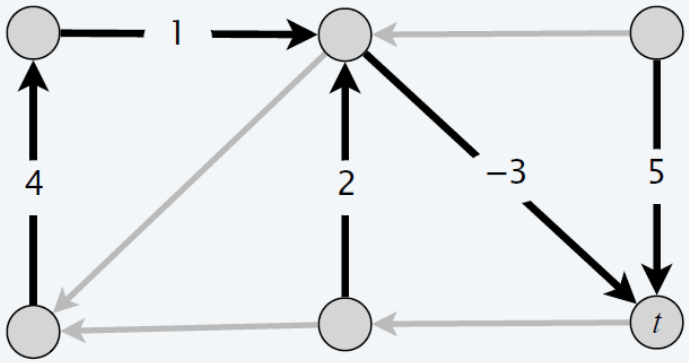


length of shortest s↝t path = 9 − 3 − 6 + 11 = 11

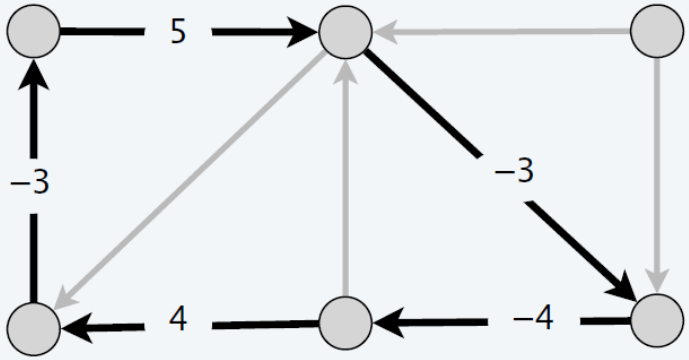
Dynamic programming: shortest path problem

Single-destination shortest-paths problem. Given a digraph $G = (V, E)$ with edge lengths ℓ_{vw} (but no negative cycles) and a distinguished node t, find a shortest $v \rightsquigarrow t$ path for every node v.

Negative-cycle problem. Given a digraph $G = (V, E)$ with edge lengths ℓ_{vw}, find a negative cycle (if one exists).



shortest-paths tree



negative cycle

Dynamic programming: shortest path problem

Def. $OPT(i, v)$ = length of shortest $v \rightsquigarrow t$ path that uses $\leq i$ edges.

Goal. $OPT(n-1, v)$ for each v.

by Lemma 2, if no negative cycles, there exists a shortest $v \rightsquigarrow t$ path that is simple

Case 1. Shortest $v \rightsquigarrow t$ path uses $\leq i-1$ edges.

- $OPT(i, v) = \text{OPT}(i-1, v)$.

optimal substructure property (proof via exchange argument)

Case 2. Shortest $v \rightsquigarrow t$ path uses exactly i edges.

- if (v, w) is first edge in shortest such $v \rightsquigarrow t$ path, incur a cost of ℓ_{vw}.
- Then, select best $w \rightsquigarrow t$ path using $\leq i-1$ edges.

Bellman equation.

$$OPT(i, v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = t \\ \infty & \text{if } i = 0 \text{ and } v \neq t \\ \min\left\{ OPT(i-1, v),\ \min_{(v,w) \in E} \{OPT(i-1, w) + \ell_{vw}\} \right\} & \text{if } i > 0 \end{cases}$$

Dynamic programming: shortest path problem

SHORTEST-PATHS(V, E, ℓ, t)

FOREACH node $v \in V$:

  $M[0, v] \leftarrow \infty$.

$M[0, t] \leftarrow 0$.

FOR i = 1 TO $n - 1$

  FOREACH node $v \in V$:

    $M[i, v] \leftarrow M[i-1, v]$.

    FOREACH edge $(v, w) \in E$:

      $M[i, v] \leftarrow \min \{ M[i, v],\ M[i-1, w] + \ell_{vw} \}$.

Theorem 1. Given a digraph $G = (V, E)$ with no negative cycles, the DP algorithm computes the length of a shortest $v \rightsquigarrow t$ path for every node v in $\Theta(mn)$ time and $\Theta(n^2)$ space.

Dynamic programming: shortest path problem

Space optimization. Maintain two 1D arrays (instead of 2D array).

- $d[v]$ = length of a shortest $v \rightsquigarrow t$ path that we have found so far.
- $successor[v]$ = next node on a $v \rightsquigarrow t$ path.

Performance optimization. If $d[w]$ was not updated in iteration $i - 1$, then no reason to consider edges entering w in iteration i.

Theorem 2. Assuming no negative cycles, Bellman–Ford–Moore computes the lengths of the shortest $v \rightsquigarrow t$ paths in $O(mn)$ time and $\Theta(n)$ extra space.

Pf. Lemma 2 + Lemma 5. ▪

shortest path exists and has at most $n-1$ edges

after i passes, $d[v] \leq$ length of shortest path that uses $\leq i$ edges

Dynamic programming: shortest path problem

BELLMAN–FORD–MOORE(V, E, c, t)

FOREACH node $v \in V$:

  $d[v] \leftarrow \infty$.

  $successor[v] \leftarrow null$.

$d[t] \leftarrow 0$.

FOR $i = 1$ TO $n - 1$

  FOREACH node $w \in V$:

    IF ($d[w]$ was updated in previous pass)

      FOREACH edge $(v, w) \in E$:

        IF ($d[v] > d[w] + \ell_{vw}$)

          $d[v] \leftarrow d[w] + \ell_{vw}$.

          $successor[v] \leftarrow w$.

  IF (no $d[\cdot]$ value changed in pass i) STOP.

pass i
$O(m)$ time

Dynamic programming: shortest path problem

Claim. ~~Throughout Bellman–Ford–Moore, following the $successor[v]$ pointers gives a directed path from v to t of length $d[v]$.~~

Counterexample. Claim is false!

- Length of successor $v \rightsquigarrow t$ path may be strictly shorter than $d[v]$.
- If negative cycle, successor graph may have directed cycles.

Lemma 6. Any directed cycle W in the successor graph is a negative cycle.

Theorem 3. Assuming no negative cycles, Bellman–Ford–Moore finds shortest $v \rightsquigarrow t$ paths for every node v in $O(mn)$ time and $\Theta(n)$ extra space.

Dynamic programming: negative cycle

Negative cycle detection problem. Given a digraph $G = (V, E)$, with edge lengths ℓ_{vw}, find a negative cycle (if one exists).

Lemma 7. If $OPT(n, v) = \text{OPT}(n-1, v)$ for every node v, then no negative cycles.

Pf. The $OPT(n, v)$ values have converged $\Rightarrow$ shortest $v \rightsquigarrow t$ path exists. ▪

Lemma 8. If $OPT(n, v) < \text{OPT}(n-1, v)$ for some node v, then (any) shortest $v \rightsquigarrow t$ path of length $\leq n$ contains a cycle W. Moreover W is a negative cycle.

Theorem 5. Can find a negative cycle in $O(mn)$ time and $O(n)$ extra space.