Problem (Best-case time)

Show how to take nearly any algorithm and modify it so it has good best-case time.

Solution

Given algorithm $\mathbb{A}$ for problem $\mathbb{P}$, we construct a new algorithm $\mathbb{A}'$ for $\mathbb{P}$ whose best-case time is "as good as possible" as follows.

For every n we have to be able to efficiently recognize an instance I of $\mathbb{P}$ of length n whose solution is trivial to compute:

<u>algorithm</u> $\mathbb{A}'(I)$ <u>begin</u>
  <u>if</u> I is a trivial instance of length $|I|$ <u>then</u>
    <u>output</u> trivial solution for I
  <u>else</u>
    <u>output</u> $\mathbb{A}(I)$
<u>end</u>.

For many $\mathbb{P}$ we can recognize a trivial instance and output its trivial solution in $\Theta(n)$ time (e.g. recognizing an already-sorted input when $\mathbb{P}$ is sorting). This would yield an algorithm $\mathbb{A}'$ with $\Theta(n)$ best-case time even when $\mathbb{A}$ is an arbitrarily poor algorithm for $\mathbb{P}$.

$\square$

Problem (Counting inversions)

Def An *inversion* in an array $A[1:n]$ is a pair of indices (i,j) such that $1 \le i < j \le n$ and $A[i] > A[j]$.

(a) Proposition The array $A = (n, n-1, \cdots, 2, 1)$ maximizes the number of inversions.

Proof Since each inversion corresponds to a pair $\{i,j\}$ of distinct indices and there are $\binom{n}{2}$ such pairs, an array can have $\le \binom{n}{2}$ inversions.
Array A above meets this upper bound, so it is optimal. $\square$

(b) Proposition Insertion sort on an n-element array with k inversions runs in $\Theta(n+k)$ time.

Analysis The structure of insertion sort is:

```
for i := 1 to n do begin
    Move A[i] to the left past all elements
    in A[1 : i-1] with value > A[i].
end
```

At step i, $A[1:i-1]$ is sorted, so the elements in $A[1:i-1]$ with value $> A[i]$ are contiguous. Thus the work at step i is $\Theta(\#\text{ elements in } A[1:i-1] \text{ with value } > A[i])$.
Summing this over all steps adds up to $\Theta(k)$.
The remaining overhead of the for-loop is $\Theta(n)$.
So the total time is $\Theta(n+k)$.

$\square$

Prob. cont'd (Counting inversions)

(c) Proposition The inversions of an n-element array can be counted in $\Theta(n \log n)$ time.

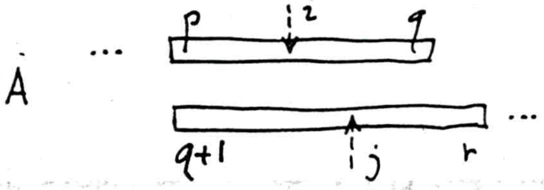
Algorithm We use divide-and-conquer, as in merge sort:

· Initially call Inversions $(A, 1, n)$.

```
function Inversions (A, p, r) begin        · Count # inversions
                                              in A[p:r].
  if p<r then begin
    q := ⌊(p+r)/2⌋
    return  Inversions (A,p,q) + Inversions (A, q+1, r)
              + SpanningInversions (A, p, q, r)
  end else
    return 0
end
```

$T(n) = 2\,T(\frac{n}{2}) + \Theta(n) = \Theta(n \log n)$

```
function SpanningInversions (A, p, q, r) begin
                                    · Count # inversions
  k := 0                              (i,j) ∈ [p,q] × (q,r].
  Merge sorted subarrays              Assumes A[p:q] and
   A[p:q] and A[q+1:r]                A[q+1:r] are sorted.
   into the sorted subarray
   A[p:r] as in merge sort:
```



Compare $A[i]$ to $A[j]$ and advance.

$\Theta(n)$ time where $n := r-p+1$

```
  Whenever A[j] is chosen from A[q+1:r],
   count # elts remaining in A[i:q] and
   accumulate this count:
                k +:= q-i+1
  return k
end
```

∎

Problem (Maximum-sum 2D subarray)

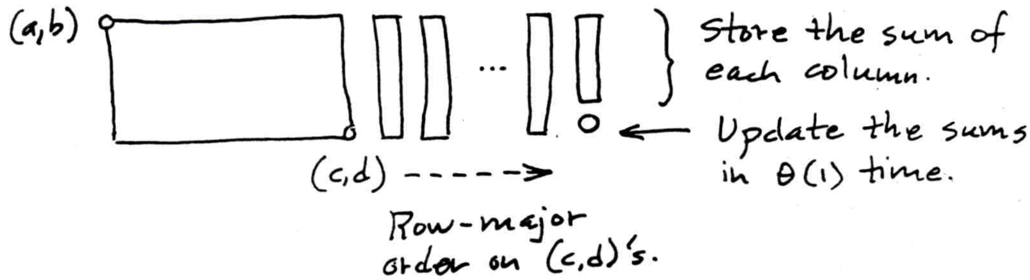
Given an $m \times n$ array $A[1:m, 1:n]$ of real numbers, find a subarray $A[a:c, b:d]$ of maximum total sum.

(a) **Proposition** Using exhaustive search, we can find a solⁿ in $\Theta(m^2 n^2)$ time using $\Theta(\min\{m,n\})$ working space.

Algorithm For a given upper-left corner (a,b), we enumerate all lower-right corners (c,d) in row-major order, storing column-sums in an array $S[1:n]$ where

$$S[j] := \sum_{a \le i \le c} A[i,j] \quad \text{for each } b \le j \le n:$$

· If $n > m$, transpose A to achieve $\Theta(\min\{m,n\})$ space.



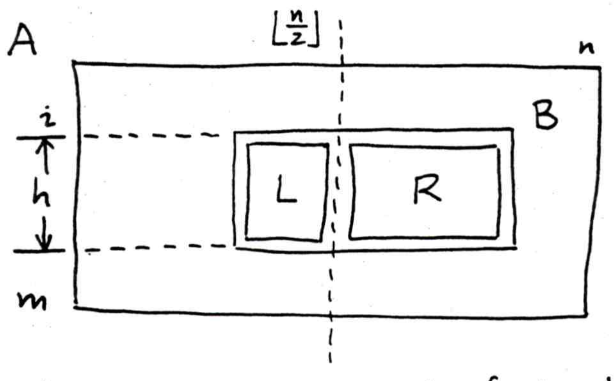

```
function Exhaustive (A, m, n) begin
  M := 0
  for a := 1 to n do
    for b := 1 to n do begin
      for j := b to n do
        S[j] := 0
      for c := a to m do begin
        T := 0
        for d := b to n do begin
          S[d] +:= A[c,d]
          T +:= S[d]               } Θ(1)
          M := max {M, T}
        end                        } Θ(n-b+1)
      end                          } Θ((m-a+1)·(n-b+1))
    end
  return M
end
```

$\sum_{1 \le a \le m} \sum_{1 \le b \le n} \Theta((m-a+1)\cdot(n-b+1)) = \Theta(m^2 n^2)$ time.

Prob. cont'd (Max-sum 2D subarray)

(b) Proposition Using divide-and-conquer, we can find a sol^n in $\Theta(m^2 n \log n)$ time using $\Theta(n)$ working space.

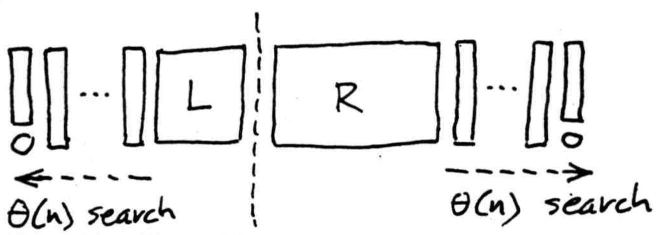
Algorithm We split the array vertically in half. The best subarray contained in each half can be found by two recursive calls. We find the best subarray spanning the split as follows:



Let (i,h) be the top row and height of the best spanning subarray B. Cutting B at the split gives two pieces L, R that must be the best subarrays with parameters (i,h) that touch the split.

Storing column-sums as in Part (a), we can find L and R independently in $\Theta(n)$ time:

• This reduces the 2D problem to $\Theta(m^2)$ 1D problems.



Enumerating all $\Theta(m^2)$ pairs (i,h) lexicographically and updating each column-sum in $\Theta(1)$ time finds the best spanning subarray in $\Theta(m^2 n)$ time.

Analysis This takes time
$$T(m,n) = 2\,T\!\left(m, \tfrac{n}{2}\right) + \Theta(m^2 n) = \Theta(m^2 n \log n).$$
□

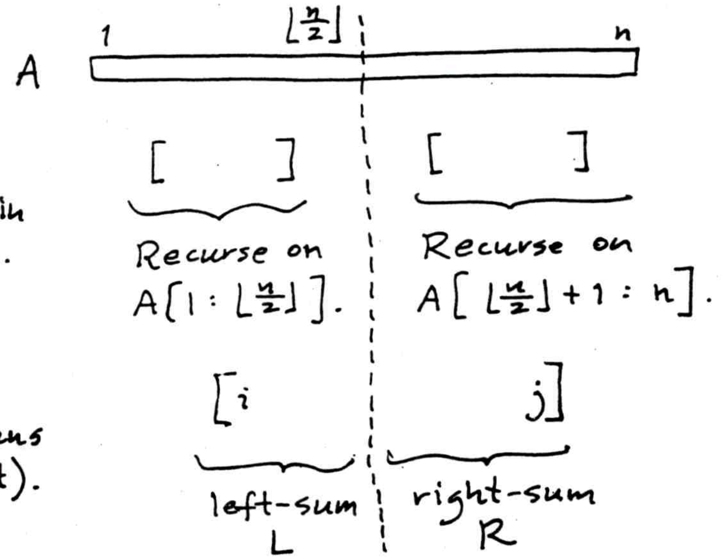
Problem (Minimum positive-sum subarray)

Given array $A[1:n]$ of real numbers, find a subarray $A[i:j]$ s.t. $\sum_{i \le k \le j} A[k]$ is

- strictly greater than zero, and
- minimum.

(a) **Proposition** Using divide-and-conquer, we can find a minimum positive-sum subarray in $\Theta(n \log^2 n)$ time.

Algorithm We split the array in half and consider where the optimal subarray might fall w.r.t. the split:



$2\,T(\frac{n}{2})$ · Cases 1,2 (sol^{n} falls in one half).

· Case 3 (sol^{n} spans the split).

For Case 3, compute all $\lceil \frac{n}{2} \rceil$ possible right-sums $\sum_{\lfloor \frac{n}{2} \rfloor < k \le j} A[k]$.

$+\,\Theta(n \log n)$ Sort these sums.

$+\,\Theta(n \log n)$ Then for each left-sum L, find the minimum right-sum R s.t. $R > -L$ using binary search on the sorted sums.

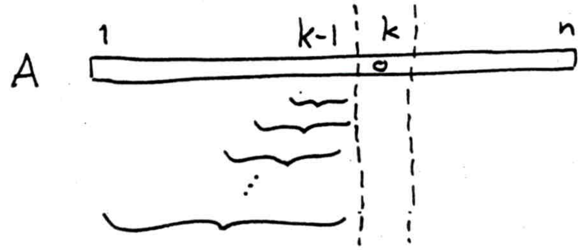
Record the best (L, R) pair.

Analysis This takes time $T(n) = 2\,T(\frac{n}{2}) + \Theta(n \log n)$
$= \Theta(n \log^2 n)$. $\square$

Prob. cont'd (Min. pos.-sum subarray)

(b) Proposition Using an incremental strategy, we can solve the problem in $O(n \log n)$ time.

Algorithm For $k = 1, 2, \cdots, n$ we find the best solution whose right end is at k, given that this problem has been solved for $k-1$:



We maintain a balanced search tree T of intervals whose right end is at $k-1$, and a real number δ. Each elt of T is a key-item pair (x, i) where $x + \delta = \sum_{i \le j < k} A[j]$. (Initially T is empty and $\delta = 0$.)

Instead of incrementing keys in T when k is increased, we just increment δ.

<u>function</u> MinPosSumSubarray (A, n) <u>begin</u>

$m := \infty$

$\delta := 0$

$T :=$ Tree()

<u>for</u> $k := 1$ to n <u>do</u> <u>begin</u>

Insert $(-\delta, k)$ into T. • Inserts the empty interval.

Find the elt (x, i) of T with smallest key x s.t. $x > -(\delta + A[k])$. • Notice $x + \delta + A[k] > 0$.

<u>if</u> elt (x, i) exists <u>then</u>

$m := \min\{m, x + \delta + A[k]\}$

$\delta \mathrel{+}:= A[k]$ • Appends $A[k]$ to all intervals in T.

<u>end</u>

<u>if</u> $m < \infty$ <u>then</u> <u>return</u> m <u>else</u> <u>return</u> $\perp$ $\square$

<u>end</u>

$O(n \log n)$ time

$O(\log n)$ time