**Problem** (Best-case time)

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

**Solution**

Given algorithm $A$ for problem $P$, we construct a new algorithm $A'$ for $P$ 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 $P$ of length $n$ whose solution is trivial to compute:

```plaintext
algorithm $A'(I)$ begin
  if $I$ is a trivial instance of length $|I|$ then
    output trivial solution for $I$
  else
    output $A(I)$
end:
```

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

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

(a) **Proposition** The array $A = (n, n-1, \ldots, 2, 1)$ 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 at most $\binom{n}{2}$ inversions. Array $A$ above meets this upper bound, so it is optimal.

(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:

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

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 $\Theta(n+k)$. □
Prop. cont. (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 $\left(A, 1, n\right)$.

\[
T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \log n)
\]

\[
\text{function } \text{Inversions} \left(A, p, r\right) \text{ begin}
\]

\[
\begin{cases}
\text{if } p < r \text{ then begin} \\
q := \left\lfloor \frac{p + r}{2} \right\rfloor \\
\text{return } \text{Inversions} \left(A, p, q\right) + \text{Inversions} \left(A, q+1, r\right) + \text{Spanning Inversions} \left(A, p, q, r\right)
\end{cases}
\]

\text{end else begin}
\text{return 0}
\text{end}
\]

\text{function } \text{Spanning Inversions} \left(A, p, q, r\right) \text{ begin}

\text{Count all inversions } (i, j) \in [p, q] \times [q, r].


\[
\Delta \quad \begin{array}{c}
0 \\
\vdots \\
q+1 \\
j \\
r
\end{array} \
\text{Compare } A[i] \text{ to } A[j] \text{ and advance.}
\]

\text{Return } k
\]

$\Theta(n)$ time where
\[
n := r - p + 1
\]
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 solution in \( \Theta(m^2n^2) \) 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

\[
S[j] := \sum A[i,j] \quad \text{for each } b \leq j \leq n:
\]

\[
(a,b) \quad (c,d) \quad \ldots \quad \text{Store the sum of each column.}
\]

\[
(c,d) \quad \text{Update the sums in } \Theta(1) \text{ time.}
\]

Row-major order on \((c,d)\)'s.

\[
\sum_{1 \leq a \leq n} \sum_{b \in \{a, \ldots, n\}} \Theta((m-a+1)(n-b+1)) \quad \sum_{1 \leq a \leq m} \sum_{b \in \{1, \ldots, n\}} \Theta((m-a+1)(n-b+1)) \quad \sum_{1 \leq a \leq m} \sum_{b \in \{1, \ldots, n\}} \Theta(m^2n^2)
\]

\[
\text{time.}
\]

function Exhaustive \((A, m, n)\) begin

\[
M := 0
\]

for \( a := 1 \) to \( n \) do

\[
\text{for } b := 1 \text{ to } n \text{ do begin}
\]

\[
\text{for } j := b \text{ to } n \text{ do begin}
\]

\[
S[j] := 0
\]

\[
\text{for } c := a \text{ to } m \text{ do begin}
\]

\[
T := 0
\]

\[
\text{for } d := b \text{ to } n \text{ do begin}
\]

\[
S[d] := A[c,d]
\]

\[
T := S[d]
\]

\[
M := \max \{M, T\}
\]

end end

end

return \( M \)
Problem cont (Max-sum 2D subarray)

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

Analysis This takes time $T(m,n) = 2T(m, \frac{n}{2}) + \Theta(m^2n) = \Theta(m^2n \log n)$. 
Problem (Minimum positive-sum subarray)


- 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:

$$2^T(n) \begin{cases} 
\text{Cases 1,2} \\
(\text{soln falls in one half).} \\
\text{Recurse on } A[1: \lfloor n/2 \rfloor]. \hspace{1cm} \text{Recurse on } A[\lfloor n/2 \rfloor + 1 : n].
\end{cases}$$

Case 3 (soln spans the split):

$$A[i] \quad A[j]$$

For Case 3, compute all $\lfloor n/2 \rfloor$ possible right-sums $\sum_{x=k}^{\lfloor n/2 \rfloor} A[x]$.

+ $\Theta(n \log n)$

Sort these sums.

Then for each left-sum $L$, find the minimum right-sum $R$ s.t. $R \geq -L$ using binary search on the sorted sums.

Record the best $(L,R)$ pair.

Analysis This takes time $T(n) = 2^T(n) + \Theta(n \log n)$

$= \Theta(n \log^2 n)$. 

$\square$
Prop. cont. (Min. pos. sum subarray)

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

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

```
A
```

We maintain a balanced search tree $T$ of intervals whose right end is at $k-1$, and a real number $S$. Each elt of $T$ is a key-item pair $(x, i)$ where

$x + S = \sum_{i : j < k} A[j]$. (Initially $T$ is empty and $S=0$.)

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

```
function MinPosSumSubarray (A, n) begin
    m := \infty
    S := 0
    T := Tree()
    for $k := 1$ to $n$ do begin
        Insert $(-S, k)$ into $T$. · Inserts the empty interval.
        Find the elt $(x, i)$ of $T$ with smallest key $x$ s.t. $x > -(S + A[k])$. · Notice $x + S + A[k] > 0$.
        if elt $(x, i)$ exists then
            m := min \{ m, \ x + S + A[k] \}
            S := S + A[k]
            Append $A[k]$ to all intervals in $T$. ·
        end
    end
    if $m < \infty$ then return $m$ else return $-$
end
```