CS545 - Design and Analysis of Algorithms Spring, 2015

Lecture Notes
Taught by: Professor John Kececioglu
Shuo Yang

---

1 Divide and Conquer

1.1 Idea

A Divide and Conquer algorithm consists of three phases:

1. Divide phase
   Split the original problem into smaller instances of same problem (usually $\Theta(1)$ subproblems) of size $\leq \alpha n$ for $\alpha < 1$.

2. Conquer phase
   Solve the subproblem recursively.

3. Combining phase
   Combine solutions of subproblems to attain the solution to the original problem.

The key to applying this technique is to discover how a solution to the input problem can be composed from (or how to decomposed into) solution of subproblems of the same form. Problem decomposition involves studying the structure of the solution.

1.2 Example

Largest Empty Rectangle

Input: 2-D boolean array $A[1:m, 1:n]$ of 0s and 1s.

Output: Find a subarray $A[i:k, j:l]$ with only 0s of largest area.

Exhaustive search(first attempt): exhaustively search for all possible rectangles that contains only 0s and find out the largest. This takes $\Theta(n^3m^3)$ time to do, very inefficient.

A divide and conquer approach:
Cut A vertically into half at the middle column. Then the optimal solution falls into three cases:

- Case-1: strictly in the left half;
- Case-2: strictly in the right half;
- Case-3: spans the middle.

We can recursively solve case-1 and case-2. For case-3, the largest empty rectangle breaks into two pieces:

- Piece-a: left rectangle ends at mid-column and is optimal conditioned on starting and ending at its first and last rows.

- Piece-b: right rectangle starts at mid-column and is optimal conditioned on starting and ending at its first and last rows.

For a given choice of row i and k, we can find the optimal left piece-a (starting at i, ending at k) by taking a min of the column-left-run-length of 0s on rows $i \cdots k$ ending at the mid-column. Similar for the optimal right piece-b.

So we precompute:
$L[i,j]$:= length of the longest run of 0s in row i that ends at (i,j).
$R[i,j]$:= length of the longest run of 0s in row i that starts at (i,j).

The following pseudo code computes L.

```
Function $PrecomputeLeftRuns(A[1:m,1:n])$
    for $i := 1$ to $m$
        $L[i,0] := 0$
        for $j := 1$ to $n$
            if $A[i,j] == 0$
                $L[i,j] := 1 + L[i,j-1]$
            else
                $L[i,j] := 0$
```

The following pseudo code computes R.

```
Function $PrecomputeRightRuns(A[1:m,1:n])$
    for $i := 1$ to $m$
        if $A[i,n] == 0$
            $R[i,n] = 1$
        else
            $R[i,n] = 0$
        for $j := n-1$ to $1$
            if $A[i,j] == 0$
                $R[i,j] := 1 + R[i,j+1]$
            else
                $R[i,j] := 0$
```

The following pseudo code computes case-3.

```
Function $SpanningRect(A, m, lo, hi, L, R)$
    $mid := \lfloor \frac{hi+lo}{2} \rfloor$
    $S := 0$ // holds the area of the largest empty rectangle spanning the mid column.
    for $i := 1$ to $m$ // row start
        $minL = \infty$
        $minR = \infty$
        for $k := i$ to $m$ // row end
            $minL = min(minL, L[k, mid])$
            $minR = min(minR, R[k, mid])$
            $S = max(S, (k-i+1) \times (minR + minL))$
    return $S$
```

The pseudo code for divide and conquer is:

```
Function $LargestEmptyRect(A, m, lo, hi, L, R)$
  if $lo < hi$
        $S := 0$
        $mid := \lfloor \frac{hi+lo}{2} \rfloor$
        $S := max(S, LargestEmptyRect(A, m, lo, mid, L, R))$
        $S := max(S, LargestEmptyRect(A, m, mid + 1, hi, L, R))$
        $S := max(S, SpanningRect(A, m, lo, hi, L, R))$
        return $S$
  else // lo == hi
        return $LR[lo]$ // the length of longest run of 0s in that column $A[1 : m, lo]$
```

The following pseudo code computes the length of longest run of 0s for each column, the result is in the table $LR[1 : n]$.

```
Function $PrecomputeLongestRuns(A[1 : m, 1 : n])$
    for $col := 1$ to $n$
        if $A[m, col] == 0$
            $prev\_run := 1$
        else
            $prev\_run := 0$
        max_run := prev_run
        for $i := m - 1$ to 1
            if $A[i, col] == 0$
                $curr\_run := 1 + prev\_run$
            else
                $curr\_run := 0$
            $max\_run = max(max\_run, curr\_run)$
            $prev\_run = curr\_run$
        $LR[col] := max\_run$
```

Putting together, the final pseudo code is:

```
Function $FindLargestEmptyRect(A, m, lo, hi, L, R)$
    $L := PrecomputeLeftRuns(A)$
    $R := PrecomputeRightRuns(A)$
    $LR := PrecomputeLongestRuns(A)$
    return $LargestEmptyRect(A, m, 1, n, L, R, LR)$
```

Run time analysis:

Subroutine $PrecomputeLeftRuns$, $PrecomputeRightRuns$ and $PrecomputeLongestRuns$ all take $\Theta(mn)$ time. Subroutine $SpanningRect$ takes $\Theta(m^2)$ time, so the recurrence equation for the routine $LargestEmptyRect$ is:

$$\begin{aligned} T(m,n) &= T(m,\frac{n}{2}) + T(m,\frac{n}{2}) + \Theta(m^2) \\ &= 2T(m,\frac{n}{2}) + \Theta(m^2) \\ &= \Theta(nm^2) \end{aligned} \tag{1}$$

The intuition behind this result is that since m does not change for n, so we can treat m as constant for n. This yields a simple recurrence equation:

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + \Theta(1) \\ &= \Theta(n) \end{aligned} \tag{2}$$

Since the algorithm always spends $\Theta(m^2)$ for each subproblem, therefore the total run time is $\Theta(n)\Theta(m^2) = \Theta(nm^2)$.

To summarize:

All possible rectangles in $A[1:m, 1:n]$ is $\Theta(m^2n^2)$. Our divide-and-conquer algorithm guarantees not to look at all possible rectangles. The key is case-3, where we first find the best left rectangle, and then the best right rectangle, then just glue them together, in doing so we can avoid searching for all possible rectangles.

Another insight is that every rectangle spans some column, so for every column j, we just ask: what is the longest rectangle spanning j, so the total run time is $\Theta(nm^2)$.

Comparison of different algorithms

Algorithm	Run time
Exhaustive search	$\Theta(m^3n^3)$ (can be reduced to $\Theta(m^2n^2)$)
Divide and conquer	$\Theta(mn \times min(m,n))$ (depends on dividing horizontally or vertically)
Dynamic programming	$\Theta(mn \times min(m,n))$
Amortized	$\Theta(mn)$ (this is optimal algorithm, have to look at every entry?)

2 Finding k_{th} smallest

2.1 Finding minimum and maximum simultaneously

algorithm using $2n-2$ comparisons

```
Function $MinMax(A[1:n])$
    $min := A[1]$
    $max := A[1]$
    for $i := 2$ to $n$
        if $A[i] < min$
            $min := A[i]$
        if $A[i] > max$
            $max := A[i]$
    return $(min, max)$
```

algorithm using $3\lfloor n/2 \rfloor$ comparisons

```
Function $MinMax(A[1:n])$
    if $n$ is odd
        $min := A[1]$
        $max := A[1]$
    else
        if $A[1] > A[2]$
            $min := A[2]$
            $max := A[1]$
        else
            $min := A[1]$
            $max := A[2]$
    while $i \leq n - 1$
        if $A[i] < A[i+1]$
            if $A[i] < min$
                $min := A[i]$
            if $A[i+1] > max$
                $max := A[i+1]$
        else
            if $A[i+1] < min$
                $min := A[i+1]$
            if $A[i] > max$
                $max := A[i]$
        $i := i + 2$
    return $(min, max)$
```

2.2 Finding k_{th} smallest in $\Theta(n)$ worst-case time

The algorithm works as follows:

```
Function $k_{th}Smallest(A[1:n], k)$
    $med := FindMedian(A[1:n])$
    $Partition(A[1:n], med)$
    if $k == \lceil \frac{n}{2} \rceil$
        return $med$
    else if $k < \lceil \frac{n}{2} \rceil$
        return $k_{th}Smallest(A[1 : \lceil \frac{n}{2} \rceil - 1], k)$
    else
        return $k_{th}Smallest(A[\lceil \frac{n}{2} \rceil + 1 : n], k - \lceil \frac{n}{2} \rceil)$
```

3 Greedy Algorithm

3.1 Activity selection problem

input: a collection of n activities, that are competing for a single resource, and must be scheduled. Activity i has a start time s_i and finish time f_i, where $s_i \leq f_i$, and it wishes to use the resource during $[s_i, f_i)$. Activities i and j are compatible if their intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.

The activity selection problem is to choose a subset of mutually compatible activities of maximum cardinality.

Greedy algorithm strategy

Let us order the activities by increasing finish time: $f_1 \leq f_2 \leq \cdots \leq f_n$. Consider the activities in this order (earliest finish time first), and add on to our subset if it is compatible with our subset. The following algorithm implements this idea where S and F are sets of start time and finish time sorted by increasing finish time, n is the number of activities.

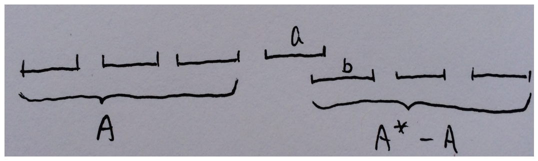
```
Function GreedyActivitySelection($S, F, n$)
    $A := \{1\}$ // earliest finished activity
    $j := 1$ // indicates the latest finish time in $A$
    for $i := 2$ to $n$
        if $S[i] \geq F[j]$
            $A := A \cup \{i\}$
            $j := i$
    return $A$
```

Correctness

Lemma: Suppose A is a subset of an optimal solution. Let a be an activity that finishes earliest among all activities that are compatible with A and that finish after the activities in A. Then $A \cup a$ is also a subset of an optimal solution.

Proof. Since A is a subset of an optimal solution, let A^* be an optimal solution with $A \subseteq A^*$. Let b be the activity in $A^* - A$ that also finishes earliest among those finish after the activities in A. We have two cases:

- case-1: $b = a$, then $A \cup \{a\} \subseteq A^*$, the lemma holds.
- case-2: $b \neq a$, then b must finish after a. We can substitute a for b in A^* and still yields an optimal solution. Thus $(A^* - \{b\}) \cup \{a\}$ is a mutually compatible set with the same cardinality as A^*, so it is also an optimal solution, and moreover, it contains $A \cup \{a\}$.



□

Theorem: *GreedyActivitySelection* finds an optimal solution.

Proof. Certainly, $\emptyset$ is always a subset of an optimal solution. Thus $\emptyset \cup \{1\}$ is a subset of an optimal solution by the lemma.

By induction on the number of iteration, when the function terminates, A is a subset of an optimal solution. And all activities not in A are incompatible with some activity in A, so A is not properly contained by any solution. Thus A is optimal. □

3.2 Proving correctness in general for greedy algorithm

When analyzing the correctness of a greedy algorithm, first prove a lemma of the following form:

Greedy Augmentation Lemma:
Suppose S is contained in an optimal solution. Let S' be an augmentation of S produced by one step of the greedy algorithm. Then S' is also contained in the optimal solution.

We can prove the lemma by *exchange-style-argument:* if the optimal solution does not contain S', then make an exchange between the augmented element in S' and an element in the optimal solution to produce another optimal solution since it did not get worse.

Notice that we need to define the containment relationship for the specific problem. It could be over set, or over string(substring, prefix, suffix, etc).

Finally, the full correctness proof of a greedy algorithm will then consist of 4 steps:

1. argue the initial state (usually $\emptyset$) is contained in an optimal solution.
2. using induction on the number of augmentations of greedy algorithm, use the lemma, argue that the output S of the greedy algorithm is contained in an optimal solution.
3. show that there exists no optimal solution that can properly contain S.