10.3
Reductions

Reduction

Reducing problem A to problem B:

1. Algorithm for A uses algorithm for B as a black box

Reduction

Reducing problem A to problem B:

1. Algorithm for A uses algorithm for B as a black box

Q: How do you hunt a blue elephant?

A: With a blue elephant gun.

Reduction

Reducing problem A to problem B:

1. Algorithm for A uses algorithm for B as a black box

Q: How do you hunt a blue elephant?

A: With a blue elephant gun.

Q: How do you hunt a red elephant?

A: Hold his trunk shut until it turns blue, and then shoot it with the blue elephant gun.

Reduction

Reducing problem A to problem B:

1. Algorithm for A uses algorithm for B as a black box

Q: How do you hunt a blue elephant?

A: With a blue elephant gun.

Q: How do you hunt a red elephant?

A: Hold his trunk shut until it turns blue, and then shoot it with the blue elephant gun.

Q: How do you shoot a white elephant?

A: Embarrass it till it becomes red. Now use your algorithm for hunting red elephants.

UNIQUENESS: Distinct Elements Problem

Problem Given an array A of n integers, are there any duplicates in A?

Naive algorithm:

```
DistinctElements(A[1..n])
    for i = 1 to n − 1 do
        for j = i + 1 to n do
            if (A[i] = A[j])
                return YES
    return NO
```

Running time: $O(n^2)$

UNIQUENESS: Distinct Elements Problem

Problem Given an array A of n integers, are there any duplicates in A?

Naive algorithm:

```
DistinctElements(A[1..n])
    for i = 1 to n − 1 do
        for j = i + 1 to n do
            if (A[i] = A[j])
                return YES
    return NO
```

Running time: $O(n^2)$

UNIQUENESS: Distinct Elements Problem

Problem Given an array A of n integers, are there any duplicates in A?

Naive algorithm:

```
DistinctElements(A[1..n])
    for i = 1 to n - 1 do
        for j = i + 1 to n do
            if (A[i] = A[j])
                return YES
    return NO
```

Running time: $O(n^2)$

UNIQUENESS: Distinct Elements Problem

Problem Given an array A of n integers, are there any duplicates in A?

Naive algorithm:

```
DistinctElements(A[1..n])
    for i = 1 to n − 1 do
        for j = i + 1 to n do
            if (A[i] = A[j])
                return YES
    return NO
```

Running time: $O(n^2)$

Reduction to Sorting

```
DistinctElements(A[1..n])
    Sort A
    for i = 1 to n − 1 do
        if (A[i] = A[i + 1]) then
            return YES
    return NO
```

Running time: $O(n)$ plus time to sort an array of n numbers

Important point: algorithm uses sorting as a black box

Advantage of naive algorithm: works for objects that cannot be "sorted". Can also consider hashing but outside scope of current course.

Reduction to Sorting

```
DistinctElements(A[1..n])
    Sort A
    for i = 1 to n − 1 do
        if (A[i] = A[i + 1]) then
            return YES
    return NO
```

Running time: $O(n)$ plus time to sort an array of n numbers

Important point: algorithm uses sorting as a black box

Advantage of naive algorithm: works for objects that cannot be "sorted". Can also consider hashing but outside scope of current course.

Reduction to Sorting

```
DistinctElements(A[1..n])
    Sort A
    for i = 1 to n − 1 do
        if (A[i] = A[i + 1]) then
            return YES
    return NO
```

Running time: $O(n)$ plus time to sort an array of n numbers

Important point: algorithm uses sorting as a black box

Advantage of naive algorithm: works for objects that cannot be "sorted". Can also consider hashing but outside scope of current course.

Two sides of Reductions

Suppose problem A reduces to problem B

1. Positive direction: Algorithm for B implies an algorithm for A
2. Negative direction: Suppose there is no "efficient" algorithm for A then it implies no efficient algorithm for B (technical condition for reduction time necessary for this)

Example: Distinct Elements reduces to Sorting in $O(n)$ time

1. An $O(n \log n)$ time algorithm for Sorting implies an $O(n \log n)$ time algorithm for Distinct Elements problem.
2. If there is no $o(n \log n)$ time algorithm for Distinct Elements problem then there is no $o(n \log n)$ time algorithm for Sorting.

Two sides of Reductions

Suppose problem A reduces to problem B

1. Positive direction: Algorithm for B implies an algorithm for A
2. Negative direction: Suppose there is no "efficient" algorithm for A then it implies no efficient algorithm for B (technical condition for reduction time necessary for this)

Example: Distinct Elements reduces to Sorting in $O(n)$ time

1. An $O(n \log n)$ time algorithm for Sorting implies an $O(n \log n)$ time algorithm for Distinct Elements problem.
2. If there is no $o(n \log n)$ time algorithm for Distinct Elements problem then there is no $o(n \log n)$ time algorithm for Sorting.

THE END

...

(for now)