Algorithm analysis illustrated: more simple examples

Given an array of numbers, what is a good way to solve each of these problems?
The solution algorithm is allowed to move the numbers around in the array.
The goal is to use a minimum number of comparisons of array entries.

max	Find the largest entry in the array.
max-min	**Find the largest entry and the smallest entry in the array.**
max-2	Find the largest entry and the second largest entry in the array.

Solution to the max-min problem

```
pair<double,double> MaxMin( double A[], int n ) {
// A is of size n > 0.  Return largest and smallest entri
  double max = A[0], min = A[0];
  for (int i = 1; i < n; ++i) {
    if (max < A[i]) max = A[i];
    else if (min > A[i]) min = A[i];
  }
  return pair<double,double>(max,min);
}
```

Is MaxMin even correct?
How many comparisons of array entries?
Is this the best one can do?

MaxMin algorithm correctness analysis

```
double max = A[0], min = A[0];
for (int i = 1; i < n; ++i) {
  if (max < A[i]) max = A[i];
  else if (min > A[i]) min = A[i];
}
```

Let's denote by $A[i..j]$, the set of entries $\{A[i], A[i+1], \ldots, A[j]\}$.

Inductive argument:
Base case: Before the loop max and min are correct for the range $A[0..0]$.
Inductive step: At the beginning of each loop iteration `max` and `min` are largest and smallest for the range $A[0..i-1]$. At the end of the iteration, this is true for the range $A[0..i]$.

Conclusion: after the loop, it is true for $A[0..n-1]$, as required.

MaxMin algorithm cost analysis

```
3   double max = A[0], min = A[0];
4   for (int i = 1; i < n; ++i) {
5     if (max < A[i]) max = A[i];
6     else if (min > A[i]) min = A[i];
7   }
```

Proposition

- *The comparison on line 5 is done $n - 1$ times.*

MaxMin algorithm cost analysis

```
3   double max = A[0], min = A[0];
4   for (int i = 1; i < n; ++i) {
5     if (max < A[i]) max = A[i];
6     else if (min > A[i]) min = A[i];
7   }
```

Proposition

- *The comparison on line 5 is done $n - 1$ times.*
- *The comparison on line 6 may be done 0 times.*

MaxMin algorithm cost analysis

```
3   double max = A[0], min = A[0];
4   for (int i = 1; i < n; ++i) {
5     if (max < A[i]) max = A[i];
6     else if (min > A[i]) min = A[i];
7   }
```

Proposition

- *The comparison on line 5 is done $n - 1$ times.*
- *The comparison on line 6 may be done 0 times.*
- *The comparison on line 6 may be done $n - 1$ times.*

MaxMin algorithm cost analysis

```
3   double max = A[0], min = A[0];
4   for (int i = 1; i < n; ++i) {
5     if (max < A[i]) max = A[i];
6     else if (min > A[i]) min = A[i];
7   }
```

Proposition

- *The comparison on line 5 is done $n - 1$ times.*
- *The comparison on line 6 may be done 0 times.*
- *The comparison on line 6 may be done $n - 1$ times.*
- *...and anywhere in between.*

MaxMin algorithm cost analysis

```
3   double max = A[0], min = A[0];
4   for (int i = 1; i < n; ++i) {
5     if (max < A[i]) max = A[i];
6     else if (min > A[i]) min = A[i];
7   }
```

Proposition

- *The comparison on line 5 is done* $n - 1$ *times.*
- *The comparison on line 6 may be done* 0 *times.*
- *The comparison on line 6 may be done* $n - 1$ *times.*
- *...and anywhere in between.*
- *MaxMin makes* $2n - 2$ *comparisons in the worst case.*

Can we do better (in the worst case)?

max-min problem optimal solution

MaxMinOpt pseudocode:

```
1 Compare in each of n/2 pairs.   // about n/2 comparisons
2 Find the max of the winners.    // about n/2 comparisons
3 Find the min of the losers.     // about n/2 comparisons
```

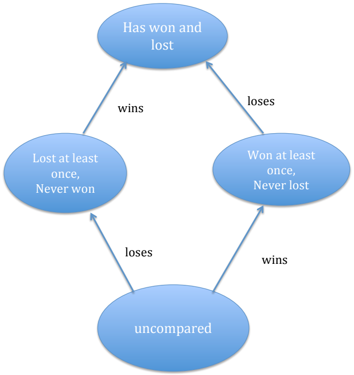
Proposition

About $3n/2$ comparisons is optimal (best possible) for the worst case number of comparisons for the max-min problem.

Proof.

(sketch) All but two elements must move from the bottom to the top of the following picture.
QED.

max-min problem optimal solution



Clicker questions

Mm(n) $=$ the problem to find the max and min of an array of n numbers.

1. Which algorithm for Mm(n) did we *not* analyze?
   - A Do the max and min independently, one after the other.
   - B Only check that an entry may be a new min if it is not a new max.
   - C Compare in pairs, then look for a max among the "winners" and for a min among the "losers".
   - D Pick two entries at random, S and L, with $S \leq L$. Look among entries less than S for the min and among entries greater than L for the max.

Clicker questions

2. Which is true?

A We found an algorithm for Mm(n) that uses about $3n/2$ entry comparisons in all cases.

B We found an algorithm for Mm(n) that uses n entry comparisons in the best case.

C We found an algorithm for Mm(n) that uses about $2n$ entry comparisons in the worst case.

D Any algorithm for Mm(n) requires at least about $3n/2$ entry comparisons in the worst case.

Clicker questions

2. Which is true?

- A We found an algorithm for Mm(n) that uses about $3n/2$ entry comparisons in all cases.
- B We found an algorithm for Mm(n) that uses n entry comparisons in the best case.
- C We found an algorithm for Mm(n) that uses about $2n$ entry comparisons in the worst case.
- D Any algorithm for Mm(n) requires at least about $3n/2$ entry comparisons in the worst case.

We'll use the term "about", as in "function f(n) is *about* g(n)" to mean f(n)/g(n) converges to 1 as n goes to ∞.
Off-by-one is an ignorable detail in Design, not a problem in Analysis, disastrous in Implementation, important to check for in Testing.

Performance analysis terminology

Being “about” right (off by a small constant) is helpful in analysis – simpler formulas are easier to work with, to remember. ... and for our purposes the details neglected are unimportant.

Actually an even sloppier measure, “big O”, is fine for most analysis. Suppose f(n) is the time taken by an algorithm of interest — worst case over all inputs of size n — and g(n) is a simple function like n or n^2 or $n\log(n)$.

We say $f(n)$ is $O(g(n))$ if $f(n)$ tends to be proportional to $g(n)$ (i.e. $f(n)/g(n) < C$, for some constant C), as n tends to ∞.

Big O and scalability

Key point: Big O is enough to predict scalability.

If my algorithm's run time is $O(g(n))$ and my algoritm takes 1 second when $n = 1000$, then my algorithm will take about $g(10000)/g(1000)$ seconds when $n = 10000$.

Example: If my run time is $O(n^2)$ and takes 2 seconds when $n = 1000$ then I expect it to take 200 seconds when $n = 10000$. I expect it to take 20000 seconds (five and a half hours) when $n = 100000$. The formula is $time(n2)/time(n1) = g(n2)/g(n1)$, so

$$time(n2) = time(n1) * g(n2)/g(n1) = 2 * 100000^2/1000^2 = 20000,$$

when $n1 = 10^3$ and $n2 = 10^5$.

Big O and proportionality

I will often use the word "proportional" when talking about the growth rate of functions. "The run time of this algorithm is proportional to n^2" means that the run time is $O(n^2)$, in other words that the runtime of the algorithm is *bounded* by $10n^2$ (or $100n^2$, or some constant times n^2), in the worst case among all inputs of size n.
Sometimes it actually is proportional, other times it has a bounding function that is proportional.

Algorithm analysis illustrated: simple examples

Given an array of numbers, what is a good way to solve each of these problems?
The solution algorithm is allowed to move the numbers around in the array.
The goal is to use a minimum number of comparisons of array entries.

- max — Find the largest entry in the array.
- max-min — Find the largest entry and the smallest entry in the array.
- max-2 — **Find the largest entry and the second largest entry in the array.**

Solution to the max-2nd problem

Not covered in lecture

```
pair<double,double> MaxMin( double A[], int n ) {
// A is of size n > 1.  Return largest and smallest entri
  double max = A[0], second = A[1];
  if (max < second) swap(max, second);
  for (int i = 2; i < n; ++i)
    if (second < A[i])
      if (max < A[i]) { second = max; max = A[1]; }
      else            { second = A[i]; }

  return pair<double,double>(max,second);
```

Is Max2nd correct?
How many comparisons of array entries?
Is this the best one can do?

Max2nd algorithm correctness analysis

```
double max = A[0], second = A[1];
if (max < second) swap(max, second);
for (int i = 2; i < n; ++i)
  if (second < A[i])
    if (max < A[i]) { second = max; max = A[1]; }
    else            { second = A[i]; }
```

Let's denote by $A[i..j]$, the set of entries $\{A[i], A[i+1], \ldots, A[j]\}$.

Inductive argument:
Base case: Before the loop max and second are correct for the range $A[0..1]$.
Inductive step: At the beginning of each loop iteration `max` and `second` are largest and second largest for the range $A[0..i-1]$. At the end of the iteration, this is true for the range $A[0..i]$.

Max2nd algorithm correctness 2

In more detail, there are three cases to be sure of:

1. A[i] $\leq$ second,
2. second ¡ A[i] $\leq$ max, and
3. max ¡ A[i].

Conclusion: after the loop, it is true for $A[0..n-1]$, as required.

```
  for (int i = 2; i < n; ++i)
    if (second < A[i])
      if (max < A[i]) { second = max; max = A[1]; }
      else            { second = A[i]; }
```

Max2nd algorithm cost analysis

```
3    double max = A[0], second = A[1];
4    if (max < second) swap(max, second);
4    for (int i = 2; i < n; ++i)
5      if (second < A[i])
6        if (max < A[i]) { second = max; max = A[1]; }
7        else            { second = A[i]; }
```

Proposition

- *The comparison on line 5 is done at most $n - 2$ times.*

Max2nd algorithm cost analysis

```
3   double max = A[0], second = A[1];
4   if (max < second) swap(max, second);
4   for (int i = 2; i < n; ++i)
5     if (second < A[i])
6       if (max < A[i]) { second = max; max = A[1]; }
7       else            { second = A[i]; }
```

Proposition

- *The comparison on line 5 is done at most $n - 2$ times.*
- *Hence, the comparison on line 6 is done at most $n - 2$ times.*

Max2nd algorithm cost analysis

```
3    double max = A[0], second = A[1];
4    if (max < second) swap(max, second);
4    for (int i = 2; i < n; ++i)
5      if (second < A[i])
6        if (max < A[i]) { second = max; max = A[1]; }
7        else            { second = A[i]; }
```

Proposition

- *The comparison on line 5 is done at most $n - 2$ times.*
- *Hence, the comparison on line 6 is done at most $n - 2$ times.*
- *Max2nd makes $2n - 2$ comparisons in the worst case.*

Max2nd algorithm cost analysis

```
3   double max = A[0], second = A[1];
4   if (max < second) swap(max, second);
4   for (int i = 2; i < n; ++i)
5     if (second < A[i])
6       if (max < A[i]) { second = max; max = A[1]; }
7       else            { second = A[i]; }
```

Proposition

- *The comparison on line 5 is done at most $n - 2$ times.*
- *Hence, the comparison on line 6 is done at most $n - 2$ times.*
- *Max2nd makes $2n - 2$ comparisons in the worst case.*

Can we do better (in the worst case)?

max-2nd problem optimal solution

MaxMinOpt pseudocode:

```
1. Organize the comparisons like March Madness:
   ... sweet 16, great 8, final 4, championship.
   // uses about $n$ comparisons ($n-1$, to be precise)
2. Hold a run-off among those who lost to the winner. // uses
   about $\lg(n)$ comparisons.
```

Proposition

MaxMinOpt uses about $n + \lg(n)$ comparisons.
About $n + \lg(n)$ comparisons is optimal (best possible) for the worst case number of comparisons for the max-2nd problem.
(MaxMin uses about $2n$ comparisons in the worst case and about $2n - \lg(n)$ on average.)

http://espn.go.com/mens-college-basketball/tournament/bracket