

Issues In Algorithm Design and Analysis

CS231 Lecture 1

September 11, 2001

Handout #4

What is CS231 About?

- Designing algorithms (recipes) to solve problems.
- Proving that an algorithm correctly solves a problem.
- Analyzing the resources (time, space) required by an algorithm.
- Exploring techniques for algorithm design and analysis
 - divide/conquer/glue and recurrence equations
 - induction and loop invariants
 - greediness
 - dynamic programming
 - probabilistic algorithms and analyses
 - amortized analysis
- Learning standard algorithms in a variety of domains.

Example: Sock Matching

● *Problem:* How to efficiently match socks from the dryer?

● *One Formalization:*

- For n pairs of socks, represent sock “colors” as integers in the range $[1..n]$. (Other representations are possible, and may lead to different algorithms.)
- Given unordered array $A[1..2n]$ containing n (not necessarily distinct) pairs of colors, permute the elements so that $A[2i - 1] = A[2i]$ for all i in $[1..n]$. (Alternatives: return new array, use lists instead.)

● *Example:*

	1	2	3	4	5	6	7	8	9	10	11	12
before: A	5	4	4	1	1	4	1	4	1	4	5	4
after: A	4	4	1	1	5	5	4	4	4	4	1	1

Sock-Matching Algorithm 1

Idea: Scan A from left to right. For each non-matching pair, swap 2nd element of pair with match for 1st element found by scanning rightward.

CLRS-style Psuedo-code:

```
1 match(A)
2   for i ← 1 to (length[A]/2) do
3     if A[2i-1] ≠ A[2i] then
4       swap(A, 2i, nextIndex(A, A[2i-1], 2i))

5 nextIndex(A, e, k)
6   for h ← k to length[A] do
7     if e = A[h] then
8       return h

9 swap(A, i, j)
10  temp ← A[i]
11  A[i] ← A[j]
12  A[j] ← temp
```

Efficiency

Want to measure the time and space resource requirements of algorithms as a function of the problem size. As shown in the following slides, there are many issues to consider when doing this.

Choosing a Barometer

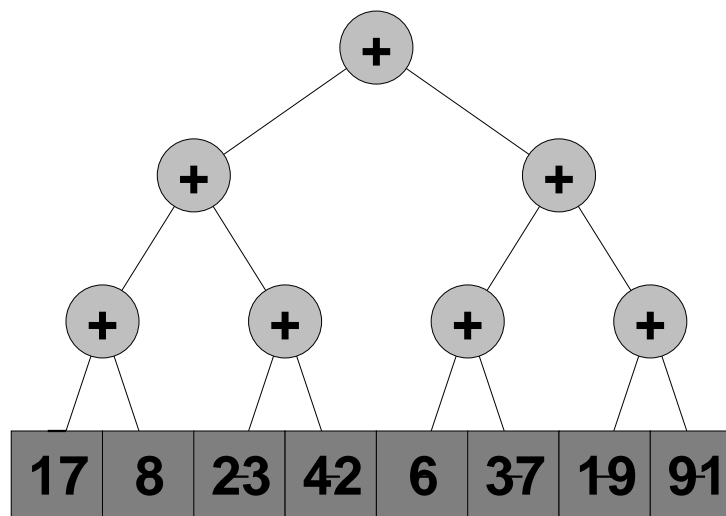
- What should we count to measure time?
 - Number of arithmetic operations (+, *, <, etc.)?
 - Number of assignments (\leftarrow) performed?
 - Number of times a line of code is executed?
- Details:
 - Some operations may be more expensive than others!
 - Do we count "hidden" increments, tests, and assignments in for loops?
 - Must pick representative line(s), usually bodies of inner loops.

Measuring Input Size

- Standard assumptions:
 - Size of numerical input is the input itself.
 - Size of array input is length of array.
- Not always obvious:
 - Size of tree may be number of nodes or height.
 - Size of number n may be 1, n , or number of bits ($\lg n$).
- Can have more than one size:
 - Searching for string of length m in text of length n .
 - Processing a graph with V vertices and E edges.
 - In sock matching, could distinguish number colors k from number of pairs n .

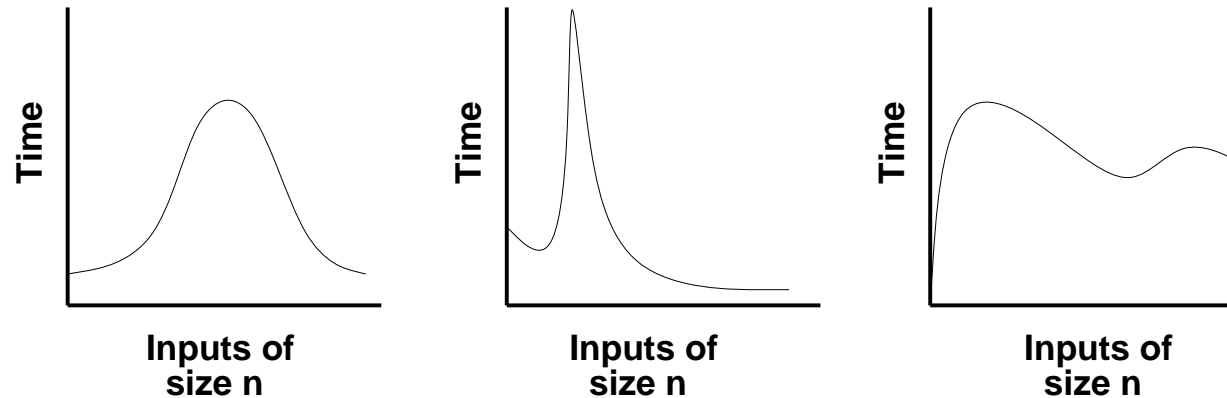
Model of Computation

- Typically assume that numerical operations take constant time.
- Addition would take linear time if model only supported increment operations.
- In practice, operations take time proportional to number of bits ($\lg n$).
- We will generally assume sequential rather than parallel model. (See CS331 for Parallel algorithms.)



Running Time for a Particular Input Size

Running time may differ greatly for different inputs of the same size.



How to characterize?

- **Best-case analysis:** consider minimum time for every input size (not a good idea!).
- **Worst-case analysis:** consider maximum time for every input size (usually the easiest analysis).
- **Average-case analysis:** expected running time based on probability distribution of inputs (can be difficult!).

Asymptotics

- Want to avoid tedious and overwhelming bean-counting.
- We will use *asymptotic notation* to abstract over details of running times. For two functions f and g :
 - $f \in o(g)$ means “ f is way smaller than g ”
 - $f \in O(g)$ means “ f is at most g ”
 - $f \in \Theta(g)$ means “ f is about the same as g ”
 - $f \in \Omega(g)$ means “ f is at least g ”
 - $f \in \omega(g)$ means “ f is way bigger than g ”
- You may be familiar with some of these from CS230; we will study all of them in the next lecture.

Analyzing Time Behavior of Algorithm 1

What is the best-case input for Algorithm 1?

What is the asymptotic best-case running time $t(n)$ for Algorithm 1?

What is the worst-case input for for Algorithm 1?

What is the asymptotic worst-case running time $t(n)$ for Algorithm 1?

Sock-Matching Algorithm 2

Idea (Bin Sorting): First scan through A , putting each sock into a bin of auxiliary array B , indexed by its color. Then scan through the bins in B , putting socks back into A .

CLRS-style Psuedo-code:

```
match(A)
1  B ← new array(length[A]/2)
2  for i ← 1 to length[B] do
3    B[i] ← 0
4  for i ← 1 to length[A] do
5    B[A[i]] ← B[A[i]] + 1
6  k ← 1
7  for i ← 1 to length[B] do
8    for j ← 1 to B[i] do
9      A[k] ← i
10     k ← k + 1
```

Analyzing Time Behavior of Algorithm 2

Does the running time of Algorithm 2 depend on the input?

What is the asymptotic running time $t(n)$ for Algorithm 2?

What About Space?

- Counting space
 - An integer variable takes one space unit.
 - A variable containing an array takes one space unit (a pointer) plus the space for the array.
 - An array of length k takes the space of k variables.
 - A function parameter counts as a variable.
 - Non-tail calls create a “call frame”; tail calls do not.
 - Typically distinguish “working space” from space for input and output.
- What is the asymptotic space $s(n)$ for Algorithm 1?
- What is the asymptotic space $s(n)$ for Algorithm 2?

Many Other Sock-Matching Algorithms

1. Probabilistic 1: Randomly pick one sock and then randomly pick another until a match is found. Repeat until done.
2. Probabilistic 2: Randomly pick *pairs* of socks until a matching pair is found. Repeat until done.
3. Mergesort-like.
4. Parallel versions of the above (i.e., multiple agents running the same code). What can go wrong?

Pop Quiz

What is the main topic of this course?

1. Quotes of the previous Vice President.
2. Patterns of growth in water flora.
3. Recipes and resources.
4. Habits of accompanying male friends.