Course Review

Jonathan Backer
backer@cs.ubc.ca

Department of Computer Science
University of British Columbia

July 23, 2007

Introduction

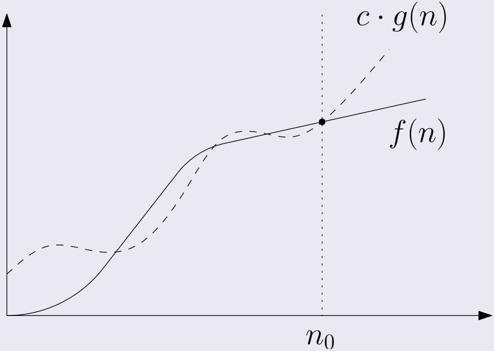
- Asymptotic notation
- Sorting
- Divide-and-conquer algorithms
- Greedy algorithms
- Dynamic programming
- Data structures
- Complexity theory

Big-O Notation

Asymptotic notation focuses on behaviour in the limit.

Definition

Let $f, g : \mathbb{N} \rightarrow \mathbb{R}$. Then $f \in O(g(n))$ if and only if $\exists c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$, $\forall n \geq n_0$.

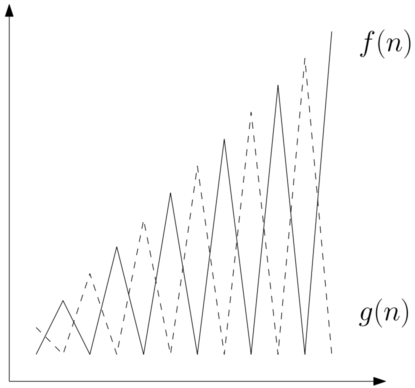


Other Asymptotic Notations

There is an intuitive correspondence:

$<$	$\leq$	$=$	$\geq$	$>$
o	O	θ	Ω	ω

Except that not every pair of functions is comparable.



Limits

Let $f, g : \mathbb{N} \to \mathbb{R}^+$. Suppose that

$$L = \lim_{n\to\infty} \frac{f(n)}{g(n)}$$

exists. Then

- if $L = 0$, then $f(n) \in o(g(n))$;
- if $L \in \mathbb{R}^+$, then $f(n) \in \Theta(g(n))$; and
- if $L = \infty$, then $f(n) \in \omega(g(n))$.

$$\begin{aligned}\lim_{n\to\infty} \frac{\sqrt{n}}{\log n} &= \frac{\infty}{\infty} \text{ so use L'Hopital's Rule} \\ &= \lim_{n\to\infty} \frac{\frac{1}{2}n^{-\frac{1}{2}}}{n^{-1}} = \lim_{n\to\infty} \frac{1}{2}\sqrt{n} = \infty\end{aligned}$$

So $\sqrt{n} \in \omega(\log n)$.

The Master Method

Theorem

Let $a \geq 1, b > 1$ be constants and $f(n) : \mathbb{N} \rightarrow \mathbb{R}^+$. Let $T(n)$ be defined by $T(n) = aT(n/b) + f(n)$, where $T(n) = \Theta(1)$ for small n and n/b is either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then

1. $T(n) \in \Theta(n^{\log_b a})$, *if* $f(n) \in O\left(n^{(\log_b a) - \epsilon}\right)$, *for some* $\epsilon > 0$.
2. $T(n) \in \Theta(n^{\log_b a} \log n)$, *if* $f(n) \in \Theta\left(n^{\log_b a}\right)$.
3. $T(n) \in \Theta(f(n))$, *if* $f(n) \in \Omega\left(n^{(\log_b a) + \epsilon}\right)$, *for some* $\epsilon > 0$, *and* $af(n/b) \leq \delta f(n)$, *for some* $\delta < 1$ *and* n *sufficiently large.*

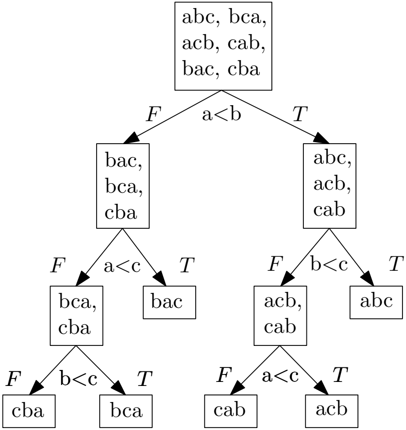
Example: Binary Search

$T(n) = T\left(\lceil n/2 \rceil\right) + 1$. Case 2 because $1 \in \Theta\left(n^{\lg 1}\right) = \Theta(n^0)$. So $T(n) \in \Theta\left(n^0 \cdot \log n\right) = \Theta(\log n)$.

Decision Tree

- Represents every sequence of comparisons that an algorithm might make on an input of size *n*.
- Nodes annotated with the orderings consistent with the comparisons made so far.
- Edges denote the result of a single comparison.
- Total order at leaves.

Algorithm: Insertion sort.
Instance ($n = 3$): the numbers a, b, c.



Bucket Sort (Counting Sort)

Assume keys are integers in ranging from 0 to $N - 1$.

- One bucket per possible key.

```
Algorithm BucketSort(A,N)
   Let S be an empty list
   Let B[0...N-1] be an array of empty lists
   for i ← 0 to A.length-1 do
      append A[i] to B[A[i].key]
   for j ← 0 to N-1 do
      for each element x of B[j] do     // in order
         append x to S
   return S
```

Time Complexity: $\Theta(n + N)$

Algorithm is stable: if `A[i].key = A[j].key` for `i < j`, then `A[i]` comes before `A[j]` in `S`

Order statistics

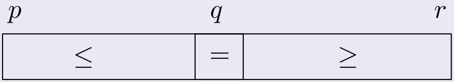
Definition

The *i*th order statistic of a set with *n* elements is the *i*th element of the set in sorted order: the 1st order statistic is the minimum and the *n*th order statistic is the maximum.

Divide-and-conquer

```
Algorithm QuickSort(A,p,r)
    if p < r then
        q ← Partition(A,p,r);
        QuickSort(A,p,q − 1)
        QuickSort(A,q + 1,r)
```

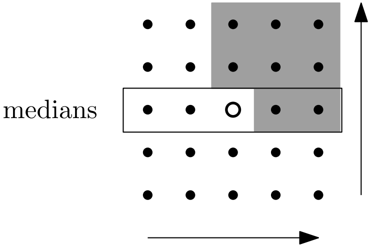
Why sort the redundant partition?



ith order statistic is in exactly one of the boxes

Order statistics (cont'd)

- A "good" pivot is close to the centre.
- A random pivot gives an average case $\Theta(n)$ solution.
  - High probability that a random pivot is good.
- The median of medians as a pivot gives a $O(n)$ solution.
  - Median of medians is a good pivot and is cheap to compute.

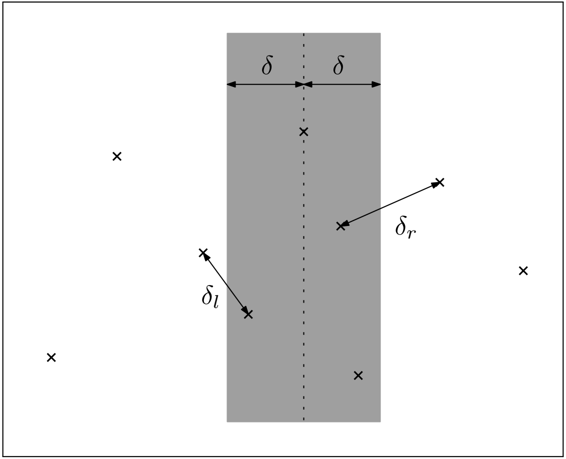


- Faster in balanced binary search trees because good pivoting is $\Theta(1)$.

Closest pair of points

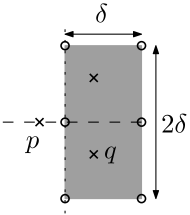
Divide-and-conquer on x-median.

- Find closest pair on left, closest pair on right, and closest pair between left and right.



Closest pair of points (cont'd)

- At most $\Theta(1)$ points on the right could be closest to one point on the left.
- Only takes $\Theta(\log n)$ to find points on the right and test them.
- So testing gray zone is just $\Theta(n \log n)$.

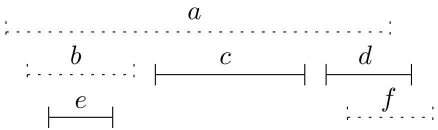


- Recurrence is $T(n) = 2T(n/2) + \Theta(n \log n)$.
- Resolves to $T(n) = n \log^2 n$ by Master Theorem.

Activity selection

Choose the next non-conflicting activity that ends the earliest to leave as much of the rest of the day available as possible.

```
Algorithm ActivitySelect(A)
    S ← ∅
    sort A by increasing right endpoints
    for j ← 1 to A.length do
        if A[j].left ≥ maxRightEndPoint(S)
            S ← S ∪ A[j]
    return S
```



We can implement this so that the comparison takes $\Theta(1)$ time.
So the algorithm runs in $\Theta(n \log n)$ time.

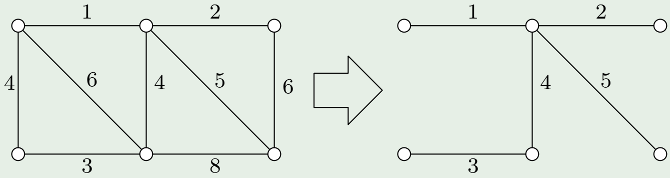
Minimum spanning trees

Problem

We are given a connected, undirected graph $G(V, E)$ with edge weights $w : E \longrightarrow \mathbb{R}^{\geq 0}$.

Find a spanning tree $T(V, E')$ with the smallest total weight $\sum_{e \in E'} w(e)$.

Example



Prim's algorithm (sketch)

- Start from a fixed vertex (v_1)
- Iteratively add the vertex that is cheapest to reach from the vertices that we have spanned so far.

```
Algorithm Prim(V, E, w)
    T ← ∅
    S ← {v₁}
    while S ≠ V do
        find e = {u, v} of minimum weight such that
             u ∈ S and v ∈ V \ S
        T ← T ∪ {e}
        S ← S ∪ {v}
    return T
```

- An $O(|V| \times |E|)$ runtime complexity as written.
- Use a priority queue (heap) to make the `find` step fast!

Single source shortest paths

Problem

Given a weighted (directed) graph G and a source vertex s, find the shortest paths from s to every other vertex of G.

Important properties:

- No vertex is visited twice on a shortest path.
- The prefix of a shortest path is a shortest path.

Outline of **Dijkstra's** algorithm:

- Grow a shortest path tree rooted at s and directed from s
- Track the cost of the shortest path to other vertices using just vertices in tree (plus the destination).
- Repeatedly add the vertex that is cheapest to reach from the tree.

Bellman-Ford algorithm

- Works with negative edge weights!
- Our first dynamic programming algorithm.
  - Divide-and-conquer breaks problems into subproblems (top-down).
  - Dynamic programming combines subproblems into problems (bottom-up).
- If there is an negative cost cycle, there is no shortest path.
- If no negative cost cycles, a shortest path visits each vertex.
  - Each shortest path uses at most $|V| - 1$ edges.
- Bellman-Ford iteratively finds shortest paths using at most $1, 2, 3, \ldots, |V| - 1$ edges.

Optimal substructure

Both Dijkstra's and Bellman-Ford's algorithms work because you can extend the optimal solution of a subproblem.

Definition

A problem has optimal substructure if some optimal solution is

- an optimal solution to a subproblem combined with
- an optimal choice.

- Often don't know which choice to make, so try them all.
- May be efficient if subproblems overlap
  - $|V|$ paths to extend in Bellman-Ford (subproblems)
  - $|E|$ edges to extend with (choices)

Making change with coins

Problem

Given: Coin values $c_1, c_2, \ldots, c_t$ with which to make change and the amount of change to be made n.

Wanted: The minimum number of coins necessary to make n cents change.

Denominations chosen so that greedy algorithm works, but not true in general.

Example

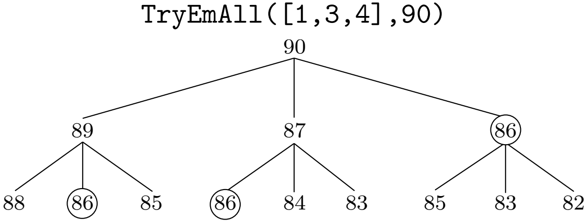
Coins: 1¢, 3¢, and 4¢	Greedy	$\longrightarrow$	4¢, 1¢, and 1¢
Change to make: 6¢	Optimal	$\longrightarrow$	3¢ and 3¢

Exhaustive coin changing

```
Algorithm TryEmAll(C,n)
    if (n = 0) then
        return 0
    int coins ← ∞
    for i ← 0 to C.length−1 do
        if (n ≥ C[i]) then
            subprob ← TryEmAll(C,n − C[i])
            coins ← min{subprob+1,coins}
    return coins
```

Recursion tree for `TryEmAll`

This is inefficient because it recomputes the same subproblems over and over again.



A better idea:

- Construct a table to store the optimal solution for each subproblem.
- Compute it recursively the first time, look it up every other time.

Memoization

```
int coins[0...n] ← ∞

Algorithm TryEmAllAgain(C,n)
    if (n = 0) then
        return 0
    if (coins[n] ≠ ∞) then
        return coins[n]
    for i ← 0 to C.length−1 do
        if (n ≥ C[i]) then
            subprob ← TryEmAll(C,n − C[i])
            coins[n] ← min{subprob+1,coins[n]}
    return coins[n]
```

Dynamic programming solution

We can eliminate the explicit recursion:

```
Algorithm DPCoinChange(C,n)
    int coins[0...n]
    coins[0] ← 0
    for nn ← 1 to n do
        coins[nn] ← ∞
        for i ← 0 to C.length−1 do
            if nn ≥ C[i] then
                coins[nn] ←
                    min{coins[nn],coins[nn − C[i]] + 1}
    return coins[n]
```

Run-time is $\Theta(nt)$.

Weighted activity selection

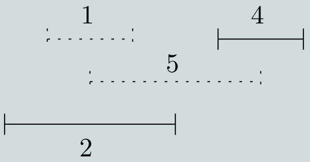
Problem

Given a set of activities represented as intervals $A = \{[a_1, b_1]], \ldots, [a_n, b_n]]\}$ and a positive weight function $w : A \rightarrow \mathbb{R}^+$ find a subset $S \subseteq A$ such that

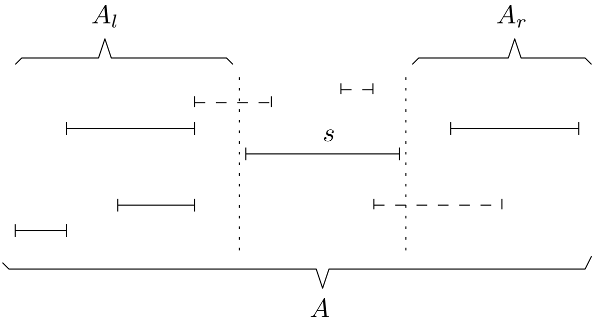
- the activities don't overlap (i.e. $s \cap t = \emptyset$, for $s, t \in S$), and
- the sum $\sum_{s \in S} w(s)$ is maximum

Example

Optimal answer is solid intervals, which simple greedy schemes don't select.



Weighted activity selection: optimal substructure



Suppose that some optimal solution S contains s

- s divides A in two: A_l completely to the left of s and A_r completely to the right of s
- $S \cap A_l$ is optimal solution to subproblem restricted to A_l
- If S_l is an optimal solution to A_l, $(S \setminus A_l) \cup S_l$ is optimal!

Weighted activity selection: divide-and-conquer

```
Algorithm MaxActivitySelect(A)
    S ← ∅
    max ← −∞
    for each [x, y] ∈ A do
        A_l ← {[a, b] ∈ A : b < x}
        A_r ← {[a, b] ∈ A : y > a}
        S_l ← MaxActivitySelect(A_l)
        S_r ← MaxActivitySelect(A_r)
        if max < ∑_{s∈S_l} w(s) + ∑_{s∈S_r} w(s) + w([x, y]) then
            S ← S_l ∪ S_r ∪ {[x, y]}
            max ← ∑_{s∈S} w(s)
    return S
```

Weighted activity select: memoization

- Cache so that we don't recompute common subproblems
- Cannot index by all subsets $A' \subseteq A$ because there are 2^n
- Showed that all subproblems were of the form $\{[a, b] \in A : \beta < a \text{ and } b < \alpha\}$ where
  - $\beta = -\infty$ or b_i and
  - $\alpha = \infty$ or a_j,

  for some i, j
- So only $O\left(n^2\right)$ subproblems...