Algorithm design techniques

```
Input:   $x$;   $e = (d_{\ell-1}d_{\ell-2} \cdots d_1d_0)_{\text{NAF}}$
Returns:   $x^e$

begin
    $S \leftarrow x$;   $R_1 \leftarrow 1$;   $R_{\bar{1}} \leftarrow 1$;
    for each digit $d_i$  ($i$ from 0 up to $\ell - 1$) do
        if $d_i \neq 0$ then
            $R_{d_i} \leftarrow R_{d_i} \times S$;
        end
        $S \leftarrow S^2$;
    end
    return $R_1 \times (R_{\bar{1}})^{-1}$;
end
```

Carlos Moreno
cmoreno@uwaterloo.ca
EIT-4103

https://ece.uwaterloo.ca/~cmoreno/ece250

These slides, the course material, and course web site are based on work by Douglas W. Harder

Algorithm design techniques

Standard reminder to set phones to silent/vibrate mode, please!

Algorithm design techniques

- During today's lesson, we'll
  - Go over some of the important algorithm design techniques, or *algorithmic paradigms*. In particular:
    - Divide-and-conquer
    - Greedy algorithms
    - Backtracking algorithms
    - Dynamic programming

Algorithm design techniques

- During today's lesson, we'll
  - Go over some of the important algorithm design techniques, or *algorithmic paradigms*. In particular:
    - Divide-and-conquer
    - Greedy algorithms
    - Backtracking algorithms
    - Dynamic programming

Algorithm design techniques

- Divide-and-conquer:
  - Break the problem into smaller sub-problems
  - Solve each of the sub-problems
  - Combine the solutions to obtain the solution to the original problem

Algorithm design techniques

- Divide-and-conquer:
  - Break the problem into smaller sub-problems
  - Solve each of the sub-problems
  - Combine the solutions to obtain the solution to the original problem
- Key detail:
  - We keep breaking the sub-problems into smaller and smaller, until the problem is transformed into something entirely different.
    - At this point, we “conquered” the original problem.

Algorithm design techniques

- Divide-and-conquer:
  - Example: In the merge sort, we have the (complicated) task of sorting a bunch of values.

Algorithm design techniques

- Divide-and-conquer:
  - Example: In the merge sort, we have the (complicated) task of sorting a bunch of values.
    - However, sorting a set of one value, however, is a *completely different* problem (a trivial one, in this case).

Algorithm design techniques

- Divide-and-conquer:
  - Example: In the merge sort, we have the (complicated) task of sorting a bunch of values.
    - However, sorting a set of one value, however, is a *completely different* problem (a trivial one, in this case).
    - Sorting a set of two values is not as trivial, but it's still a completely different problem — one that can be described as one conditional swap.

Algorithm design techniques

- Divide-and-conquer:
  - An interesting example that we have not covered earlier in the course: Karatsuba multiplication algorithm:
    - Useful for multiplying polynomials and integer number (based on a binary representation)
  - Originally proposed as an efficient technique to implement binary multiplication in hardware (i.e., an efficient design of the multiplication circuitry in an ALU)

Algorithm design techniques

- Divide-and-conquer:
  - The key detail in this case being: multiplying two numbers of *n* bits is a complicated problem — but multiplying two single-bit numbers is just a logical AND gate.

Algorithm design techniques

- Divide-and-conquer:
  - The key detail in this case being: multiplying two numbers of *n* bits is a complicated problem — but multiplying two single-bit numbers is just a logical AND gate.
  - The algorithm is nowadays used in cryptographic applications (among others), where arithmetic with very large numbers (in the order of up to thousands of bits) is required:

Algorithm design techniques

- Divide-and-conquer:
  - The key detail in this case being: multiplying two numbers of *n* bits is a complicated problem — but multiplying two single-bit numbers is just a logical AND gate.
  - The algorithm is nowadays used in cryptographic applications (among others), where arithmetic with very large numbers (in the order of up to thousands of bits) is required:
    - In this context, the difference being: multiply a number of n bits where n > CPU register width vs. multiply two register-size numbers.

Algorithm design techniques

- Divide-and-conquer:
  - The idea is that instead of the “schoolbook” approach to multiply (requiring O(n^2)), we break the number (or the polynomial) into two halves, and use a trick to express the result in terms of three multiplications of half-size operands:
    - Example shown in the board (feel free to take notes if you want, but I'll post an expanded version of these slides within the next few days)

Algorithm design techniques

- Karatsuba multiplication algorithm:
  - Say that we have to multiply two n-bit numbers A and B (to simplify the example, we'll assume that n is a power of 2, so that we can always divide into two)
  - The result is clearly a $2n$ bits number
    - Can not be more than $2n$ bits, and can require up to $2n$ bits to represent.
    - n bits can represent up to 2^n-1, and $(2^n-1) \times (2^n-1)$ is $2^{2n} - 2^{n+1} + 1 < 2^{2n}$ and thus takes $2n$ bits to represent.

Algorithm design techniques

- Karatsuba multiplication algorithm:
  - We split each of the two numbers into two halves, A_H, A_L, B_H, and B_L. Clearly, we have that:

$$A = A_H 2^{n/2} + A_L$$
$$B = B_H 2^{n/2} + B_L$$

Algorithm design techniques

- Karatsuba multiplication algorithm:
  - We split each of the two numbers into two halves, A_H, A_L, B_H, and B_L. Clearly, we have that:

$$A = A_H 2^{n/2} + A_L$$
$$B = B_H 2^{n/2} + B_L$$

Thus:

$$A \cdot B \;=\; A_H B_H 2^n + (A_H B_L + A_L B_H) 2^{n/2} + A_L B_L$$

Algorithm design techniques

- Karatsuba multiplication algorithm:
  - At first glance, the equation looks like we need four multiplications of half-size (and if we do the math, we quickly realize that that leads to quadratic run time).
  - The clever detail proposed by Karatsuba was that the operations can be re-arranged to trade one multiplication by a few additions (but additions are inexpensive — they're linear time!)

Algorithm design techniques

- Karatsuba multiplication algorithm:
  - From this equation:

$$A \cdot B = \underbrace{A_H B_H}_{D_2} 2^n + \big(\underbrace{A_H B_L + A_L B_H}_{D_1}\big) 2^{n/2} + \underbrace{A_L B_L}_{D_0}$$

  - We notice that:

$$(A_H + A_L) \cdot (B_H + B_L) = A_H B_H + A_L B_L + A_H B_L + A_L B_H$$

Algorithm design techniques

- Karatsuba multiplication algorithm:
  - From this equation:

$$A \cdot B = \underbrace{A_H B_H}_{D_2} 2^n + \underbrace{\left(A_H B_L + A_L B_H\right)}_{D_1} 2^{n/2} + \underbrace{A_L B_L}_{D_0}$$

  - We notice that:

$$(A_H + A_L) \cdot (B_H + B_L) = A_H B_H + A_L B_L + A_H B_L + A_L B_H$$

  - The left term involves one multiplication of half-size operands, plus two additions, and the result is the term D_1 with two additional terms — but these additional terms are D_0 and D_2, which can be reused, since they need to be computed anyway!

Algorithm design techniques

- Karatsuba multiplication algorithm:
  - Summarizing, we compute the following three multiplications of half-size operands:

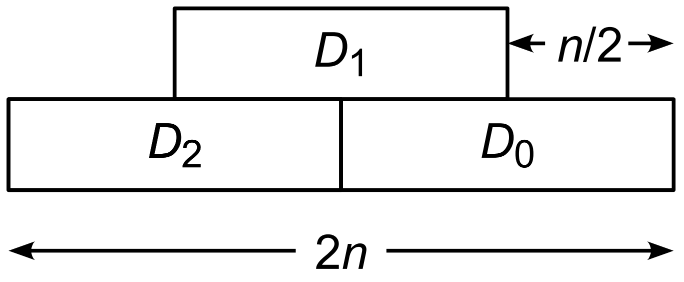
$$D_0 = A_L B_L$$
$$D_2 = A_H + B_H$$
$$D_1 = \left(A_H + A_L\right)\left(B_H + B_L\right) - D_0 - D_2$$

  - And the result is obtained as:

$$A \cdot B \;=\; D_2 2^n + D_1 2^{n/2} + D_0$$

Algorithm design techniques

- Karatsuba multiplication algorithm:
  - We recall that multiplying times a power of 2 simply means left-shifting the bits; so, the expression $D_2 2^n + D_1 2^{n/2} + D_0$ is simply obtained by adding the values at the appropriate position:



Algorithm design techniques

- Divide-and-conquer:
  - The run time is given by the following recurrence relation:

$$\mathrm{T}(n) = 3\,\mathrm{T}(n/2) + \Theta(n)$$

Algorithm design techniques

- Divide-and-conquer:
  - The run time is given by the following recurrence relation:

$$\mathrm{T}(n) = 3\,\mathrm{T}(n/2) + \Theta(n)$$

  - With this, the run time comes down to $O(3^{\lg n}) = O(n^{\lg 3}) \approx O(n^{1.585})$ (sub-quadratic time)

Algorithm design techniques

- Next, let's take a look at Greedy algorithms...

Algorithm design techniques

- These are iterative algorithms that at each iteration the criterion used is to maximize some “gain” or some objective function.

Algorithm design techniques

- These are iterative algorithms that at each iteration the criterion used is to maximize some “gain” or some objective function.
  - The term “greedy” refers to the fact that the algorithms do this in a “short sighted” way; they try to maximize immediate gain, disregarding the big picture (“get the most I can get *now*”).

Algorithm design techniques

- These are iterative algorithms that at each iteration the criterion used is to maximize some "gain" or some objective function.
  - The term "greedy" refers to the fact that the algorithms do this in a "short sighted" way; they try to maximize immediate gain, disregarding the big picture ("get the most I can get *now*").
    - For this reason, they can fail to determine a global maximum or minimum (they could "fall in a trap" and converge to some local maximum)
    - Example: finding the maximum of a function going by steps of fixed size. (example drawn in the board)

Algorithm design techniques

- Now, they don't necessarily fail — in fact, we saw our fair share of greedy algorithms that are proven to obtain the correct output ... Anyone?
  - Hint: we saw three of these very recently ...

Algorithm design techniques

- Now, they don't necessarily fail — in fact, we saw our fair share of greedy algorithms that are proven to obtain the correct output ... Anyone?
  - Hint: we saw three of these very recently ...
- Dijkstra's, Prim's, and Kruskal's algorithms are all greedy algorithms (though we'll see that Dijkstra's and Prim's also exhibit aspects of dynamic programming)

Algorithm design techniques

- Now, they don't necessarily fail — in fact, we saw our fair share of greedy algorithms that are proven to obtain the correct output ... Anyone?
  - Hint: we saw three of these very recently ...

- Dijkstra's, Prim's, and Kruskal's algorithms are all greedy algorithms (though we'll see that Dijkstra's and Prim's also exhibit aspects of dynamic programming)
  - And they *do* work (we proved some of the related details!)

Algorithm design techniques

- A classical example of this technique is the problem of producing change — or in general, expressing a given amount of money in coins, minimizing the number of coins required.
  - What's a greedy algorithm to compute this result?

Algorithm design techniques

- A classical example of this technique is the problem of producing change — or in general, expressing a given amount of money in coins, minimizing the number of coins required.
  - What's a greedy algorithm to compute this result?
  - At each iteration, put as many coins of the highest denomination that fits (so that *at this iteration* we minimize the number of coins used).

Algorithm design techniques

- A classical example of this technique is the problem of producing change — or in general, expressing a given amount of money in coins, minimizing the number of coins required.
  - What's a greedy algorithm to compute this result?
  - At each iteration, put as many coins of the highest denomination that fits (so that *at this iteration* we minimize the number of coins used).
    - But.... Does it minimize the number of coins in the final outcome of the algorithm?

Algorithm design techniques

- You're probably used to the idea that it does (because with Canadian coins / denominations, it *does* work)

Algorithm design techniques

- You're probably used to the idea that it does (because with Canadian coins / denominations, it *does* work)
- But it can indeed fail.... Anyone?

Algorithm design techniques

- You're probably used to the idea that it does (because with Canadian coins / denominations, it *does* work)
- But it can indeed fail. Consider denominations 10, 8, and 1; and try to break 17 into those.
  - The optimal is obviously 3 coins (2×8 + 1); but the greedy algorithm would tell us that we require 8 coins! (10 + 7×1)

Algorithm design techniques

- This greedy algorithm can work (as in, produce always the correct output), depending on the set of denominations:
  - If every denomination is at least twice the next one ("next one" in decreasing order, that is).

Algorithm design techniques

- The greedy algorithm can work (as in, produce always the correct output), depending on the set of denominations:
  - If every denomination is at least twice the next one ("next one" in decreasing order, that is).
  - Alternatively, if every denomination is greater than or equal to the sum of all smaller denominations.
    - Canadian coins meet these requirements; in cents, we have: 200, 100, 25, 10, 5, and 1

Algorithm design techniques

- So, if these greedy / short-sighted algorithms can fail, why do we bother? Why are they attractive?

Algorithm design techniques

- So, if these greedy / short-sighted algorithms can fail, why do we bother? Why are they attractive?
  - They tend to be *efficient* — at each iteration, we look only at the immediate scenario, no need to look globally.
  - If we can find a greedy algorithm that works, then that's good news!

Algorithm design techniques

- So, if these greedy / short-sighted algorithms can fail, why do we bother? Why are they attractive?
  - They tend to be *efficient* — at each iteration, we look only at the immediate scenario, no need to look globally.
  - If we can find a greedy algorithm that works, then that's good news!
    - On the other hand, it could be an example of a solution to a problem like an NP-complete problem, or a problem believed intractable, and a greedy algorithm might be an efficient alternative that finds a *near-optimal* solution.

Algorithm design techniques

- Next, we'll look at Backtracking algorithms...

Algorithm design techniques

- Backtracking algorithms can be seen as an optimized form of exhaustive search algorithms:
  - Try all possible combinations of things, *with early elimination* of paths that lead to no solution.

Algorithm design techniques

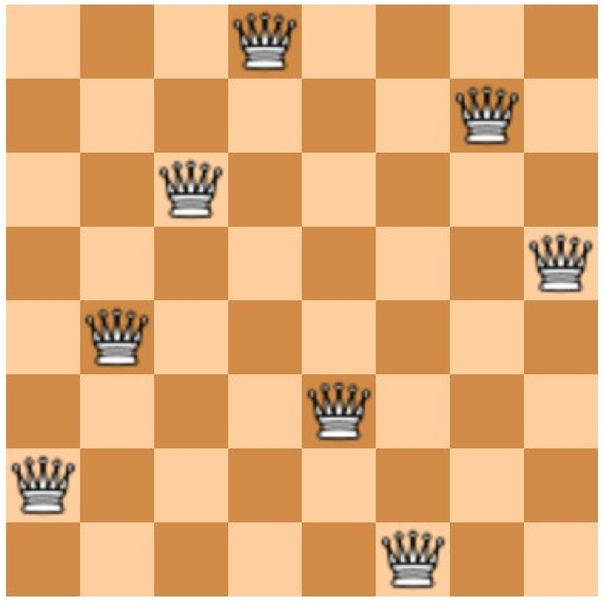
- Backtracking algorithms can be seen as an optimized form of exhaustive search algorithms:
  - Try all possible combinations of things, *with early elimination* of paths that lead to no solution.
  - A good visual analogy is: Think of the possibilities represented by a tree, and we do a depth-first traversal, but we have the advantage that in many cases, before reaching the leaf node, we'll know that this path will lead us nowhere, so we just “backtrack” and try the next possibility at the previous level.

Algorithm design techniques

- One classical example is the famous eight queens Chess puzzle:
  - Find a placement for eight queens in a chess board such that neither one falls under threat of capture.

Algorithm design techniques

- One classical example is the famous eight queens Chess puzzle:
  - Find a placement for eight queens in a chess board such that neither one falls under threat of capture.



Algorithm design techniques

- A backtracking algorithm places one queen at a time (say, column by column).
    - At each column (each “level” of the search), we have eight possibilities, so we try one and move to the next level.
        - If at some level we find that all eight possibilities fail, then we *backtrack* — go back to the previous level, reporting that this one failed; at the previous level, we now move to the next possibility and keep trying.

Algorithm design techniques

- A backtracking algorithm places one queen at a time (say, column by column).
  - At each column (each “level” of the search), we have eight possibilities, so we try one and move to the next level.
    - If at some level we find that all eight possibilities fail, then we *backtrack* — go back to the previous level, reporting that this one failed; at the previous level, we now move to the next possibility and keep trying.
    - The key detail is that most of the time, we're going to discover *early* that the current path won't work (e.g., at the third or fourth column we'll find that it won't work)

Algorithm design techniques

- Another classical example is the Turnpike reconstruction problem:
  - We have *n* points in a line, and we're given the distances (say, in ascending order) between every pair of points; that is, we're given $|x_k - x_m| \quad \forall\ k, m$
  - The problem is: find the values (the points), assuming a fixed reference point (e.g., $x_1 = 0$)
    - Notice that for *n* points, we'll have *n*(*n*−1) / 2 distances.

Algorithm design techniques

- An algorithm goes by processing the distances from highest to lowest. Let's work through this example (from Mark Allen Weiss textbook) of 6 points (6 values), x_1 to x_6 for which we're given these 15 distances:

$$\{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 10\}$$

Algorithm design techniques

- Key detail here: at each iteration, we have several possibilities, but some possibilities don't work because they lead to points with distances that are not found in the set.
  - So, we *backtrack* and discard the “current” possibility being considered at the previous level.

Algorithm design techniques

- Key detail here: at each iteration, we have several possibilities, but some possibilities don't work because they lead to points with distances that are not found in the set.
  - So, we *backtrack* and discard the “current” possibility being considered at the previous level.

- If possible, see Mark Allen Weiss book for an algorithm (actually, a C++ fragment of code) implementing this approach.

Algorithm design techniques

- Last, we'll take a look at Dynamic programming.

Algorithm design techniques

- The basic idea with Dynamic programming is to work with a problem that is broken into slightly smaller sub-problems, where some of these sub-problems overlap.
  - A straightforward recursive solution is inefficient since we solve the same sub-problems over and over again.

Algorithm design techniques

- The basic idea with Dynamic programming is to work with a problem that is broken into slightly smaller sub-problems, where some of these sub-problems overlap.
  - A straightforward recursive solution is inefficient since we solve the same sub-problems over and over again.
- The main detail with a dynamic programming solution is that we store the solutions for sub-problems that we already solved!

Algorithm design techniques

- A classical example is that of a recursive implementation of a function to determine a Fibonacci number — which can be “neatly” implemented as:

```
int fibonacci (int n)
{
    if (n <= 2)
    {
        return 1;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Algorithm design techniques

- Why is that function horribly inefficient ?
  - Computing F(6) involves computing F(5) and F(4); but then, computing F(5) requires computing F(4) and F(3), so we end up doing redundant computations.
    - Hopefully you see the exponential nature of the redundancy?

Algorithm design techniques

- Why is that function horribly inefficient ?
  - Computing F(6) involves computing F(5) and F(4); but then, computing F(5) requires computing F(4) and F(3), so we end up doing redundant computations.
    - Hopefully you see the exponential nature of the redundancy?
    - In the above example, it might seem like we're doing no more than twice as much work — but F(3), for example, is needed (indirectly) by F(6), both through the paths F(5) and F(4) — but then F(4) is being redundant.... Each extra level multiplies times something the number of times the lower F(k) are computed

Algorithm design techniques

- Why is that function horribly inefficient ?
  - There's actually a far more neat way to see that the run time is exponential.
    - The recurrence relation for that recursive function is:

$$\mathrm{T}(n) = \mathrm{T}(n-1) + \mathrm{T}(n-2) + \Theta(1)$$

    - But that's the same recurrence relation as for the Fibonacci numbers themselves, with an additional term added (the Θ(1) term).
      - Thus, T(*n*) ≥ F(*n*) — T(*n*) grows at least as fast as the sequence of Fibonacci numbers, which is known to grow exponentially!

Algorithm design techniques

- The Dynamic programming approach is, then, storing the computed values so that we don't need to redundantly compute them over and over.

Algorithm design techniques

- The Dynamic programming approach is, then, storing the computed values so that we don't need to redundantly compute them over and over.
  - Notice that we do not need an exponential amount of storage — the amount of storage is linear.

Algorithm design techniques

- The Dynamic programming approach is, then, storing the computed values so that we don't need to redundantly compute them over and over.
  - Notice that we do not need an exponential amount of storage — the amount of storage is linear.
  - The amount of *redundancy* was exponential (it's not like we compute an exponential number of values; no, we compute a small number of values *repeatedly* — for a total exponential number of computations).

Algorithm design techniques

- Two approaches:
  - Top to bottom
    - Set up an associated array where each time that the function is requested to compute one value, it first checks if that value is in the corresponding location in the array:
      - If it is, use the value
      - If it's not, compute the value and store the obtained result in the array.

Algorithm design techniques

- Two approaches:
  - Top to bottom
    - Set up an associated array where each time that the function is requested to compute one value, it first checks if that value is in the corresponding location in the array:
      - If it is, use the value
      - If it's not, compute the value and store the obtained result in the array.
  - Bottom to top
    - Start at the “base case”, and explicitly go up, calculating the “upper” values in terms of the already-calculated “bottom” values.

Algorithm design techniques

- There are many applications. Examples are:
  - Finding the optimal order of multiplications for a sequence of matrices of different sizes
    - Matrix multiplication is associative; obtaining ABCD can be done as A(BCD), or (AB)(CD), or A(BC)D, etc.
    - When considering all possible arrangements, some of the arrangements for smaller sequences repeat, so we would end up redundantly doing these computations (see for example Prof. Harder's slides for a complete example with more details)

Algorithm design techniques

- There are many applications. Examples are:
  - Finding common substrings, or aligning strings where differences are small omissions on either side.
  - Alignment by stretching sub-sequences:
    - This has very interesting applications in speech recognition and in rhythm recognition/matching for music search applications.

Summary

- During today's lesson, we:
  - Introduced four important algorithm design techniques; namely:
    - Divide-and-conquer
    - Greedy algorithms
    - Backtracking algorithms
    - Dynamic programming
  - Discussed some illustrative/representative examples of each of the techniques.
  - Discussed some of the advantages and when the given technique is appropriate.