Final Exam Review

CSCI 3100

Recurrence Relations

- How many times will line 6 get executed if x = 5, n = 0?
- How many times will line 6 get executed if x = 6, n = 0?
- How many times will line 6 get executed if n = 1?
- How many times will line 6 get executed if n > 1?

```
1. int pow(int x, int n)
2. {
3.     if (n == 0)
4.         return 1;
5.     else
6.         return x*pow(x, n-1)
7. }
```

Which recurrence relation describes the run-time of this algorithm?

```
1. int pow(int x, int n)
2. {
3.     if (n == 0)
4.         return 1;
5.     else
6.         return x*pow(x, n-1)
7. }
```

A. F(n) = F(n-1) + C
B. F(n) = 2F(n-1)+1
C. F(n) = 2^n
D. F(n) = n^2
E. F(n) = n-1

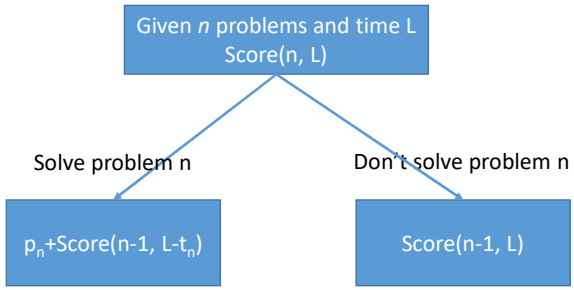
What is the asymptotic complexity of the 'pow' function from previous slide?

- F(n) = F(n-1) + C (assume F(0) = D)
- F(1) = F(0)+C = D + C
- F(2) = F(1) + C = (D + C)+C = D + 2C
- F(3) = F(2) + C = (D+2C) + C = D + 3C
- …
- F(n) = F(n-1) + C =
  F(n-2) + C + C =
  F(n-3) + C + C + C =
  …
  F(n-k) + kC
- Let k = n. Then F(n) = F(0) + kC = D + nC

A.Θ (n^2)
B.Θ (n log(n))
C.Θ (n)
D.Θ (1)
E.Θ (2^n)

Recurrences and Dynamic Programming

- Solution is defined in terms of recurrence relation (sub-problems)
- Some sub-problems are overlapping
- Example: The "final exam" problem
  - You are looking at your final exam
  - Each problem has a point value
  - You estimate how long each problem will take you
  - You have a time limit L
  - p – array of point values
  - t – array of time estimates
  - Goal: solve as many problems as possible within the time limit, while maximizing your score (sum of point values of solved problems)



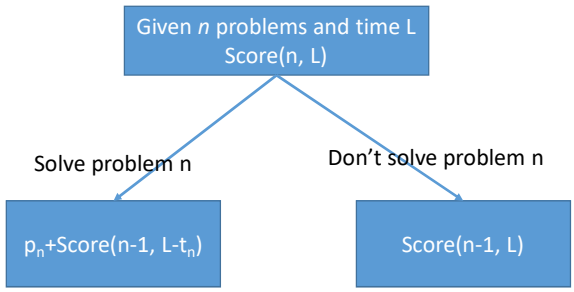
What recurrence relation describes optimal substructure of the "final exam" problem?

A. Score(n, L) = max[Score(n-1, L), p_n + Score(n-1, L-t_n)]

B. Score(n, L) = max[Score(n-1, L-t_n), p_n + Score(n-1, L)]

C. Score(n, L) = max[Score(n, L-t_n), p_n + Score(n-1, L)]

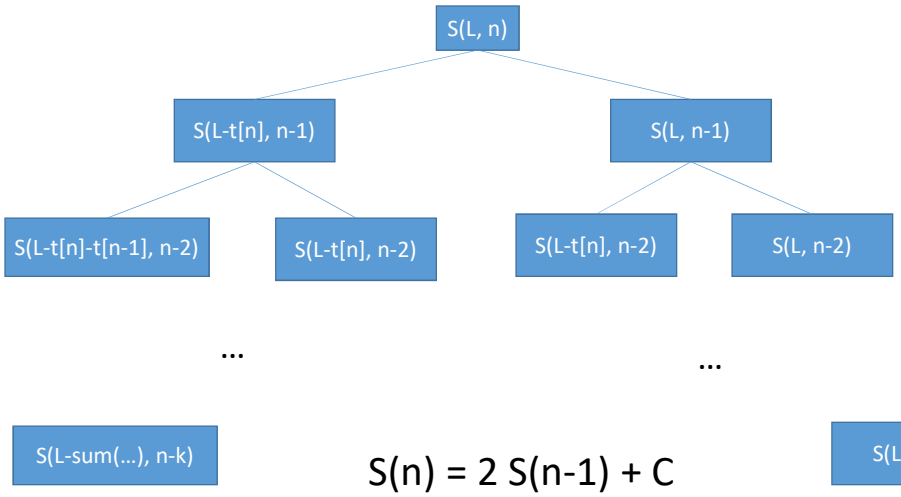
D. Score(n, L) = max[Score(n-1, L-t_{n-1}), p_n+Score(n-1, L-t_n)]



Recursive Solution – top down

```
int score(p[ ], t[ ], L, n)
{
    if (n == 1)
    { return … }
    else
    {
     return max(p[n] + score(p, t, L-t[n], n-1),
                score(p, t, L, n-1));
    }
}
```

What is the complexity of the recursive solution?



S(n) = 2 S(n-1) + C

A. $O(n^2)$
B. $O(2^n)$
C. O(n log n)
D. $O(n^4)$

Dynamic programming – bottom up

```
int score(p[ ], t[ ], L, n)
{
   int S[][]; //n by L array indexing starts at 1
   // initialize S[1]
   for (int i = 2; i <= n; i++)
   {
      for (int k = 1; k <=L; k++)
      {
         S[i][k] = max(p[i] + S[i-1][k-t[i]], S[i-1][k]);
      }
   }
}
```

What is the complexity of the dynamic programming solution?

Can we use a "greedy" algorithm for this problem?

- Potential greedy strategies:
  - Sort by points in descending order:
    - Counterexample: P = [10, 5, 5, 5], t =[60, 20, 20, 20], L = 60
  - Sort by time in ascending order:
    - Counterexample: P = [1, 1, 1, 10], t = [20, 20, 20, 60], L = 60
  - Sort by points – time in descending order:
    - Counterexample: P = [10, 20], t = [9, 20], L = 20
  - Sort by points/time in descending order:
    - Counterexample: P = [10, 11], t = [9, 10], L = 10

Greedy Algorithms we Covered

Activity Selection

- Given a set of activities with start and finish times, find the largest subset of non-overlapping activities.
- Greedy strategy – select next compatible activity with earliest finish time
- Proof: show that current greedy choice can be in an optimal solution
- By contradiction: assume that activity *m* with earliest finish time is NOT in optimal solution. Let activity *k* be an activity from the optimal solution with earliest (among all activities in the optimal solution) finish time. Swap *m* with *k*, because *m* is compatible with the rest of the activities in the optimal solution. This is another optimal solution. Therefore, our assumption was incorrect.
- Your turn: explain to your neighbor why swapping *m* for *k* results in an optimal solution

Other 'Activity' Greedy Algorithms

- Activity Partitioning
- Minimize Schedule delay

Same strategy does NOT work for Weighted Activity Selection

- Given a set of activities with start time, finish time, and weight, select a subset of non-overlapping activities such that the total weight is maximized.
- Find optimal substructure:
  - Given n activities, you can include activity n or not
  - If you include n, then your solution will have the weight:
    - W[n] + [the weight of the solution for n-1 activities]
  - If you don't include n, then your solution will have the weight:
    - [the weight of the solution for n-1 activities]
  - Pick the better of the two options
  - Write a recurrence relation
  - Turn it into "bottom-up" solution, if overlapping sub-problems exist

Minimum Spanning Trees

- Greedy property – use next smallest edge
  - Kruskal's – select next smallest edge that doesn't form a cycle
  - Prim's – select next smallest edge adjacent to current MST without forming a cycle
- Proofs: show that the current greedy choice can be included in the optimal solution.
  - Cut property
  - Cycle property

Kruskal's Algorithm

- Cut property:
  - Let G=(V,E) be an undirected connected graph
  - Let X be a subset of V. Then V-X is the set of vertices in V that are not in X.
  - Let (u,v) be an edge where u is in X and v is in G-X. So (u,v) connects X to G-X.
  - Let (u,v) be the smallest edge connecting X to G-X.
  - Claim: (u, v) is in the minimum spanning tree of G
- How would you prove the cut property?
  - A. Suppose (u, v) is not in the minimum spanning tree of G …
  - B. Suppose (u, v) is in the minimum spanning tree of G …
  - C. Show an example of a G, X, V, and (u, v) and show a MST of G that has (u, v) in it
  - D. Let G be a tree (there is a unique path from every vertex to every other vertex). Then G is a MST and (u, v) is in the MST.

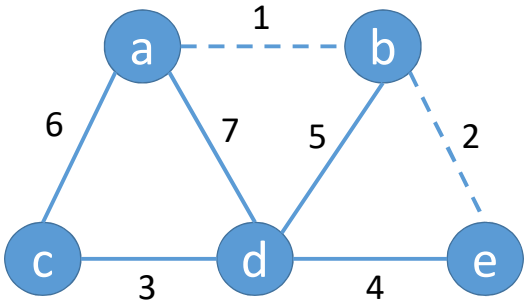
Prim's Algorithm

```
1. MST_PRIM(V, E)
2. V' = empty, E' = E
3. Select u from V, add u to V'
4. While |V'| < |V|
5.     select smallest edge (x,y) crossing the
cut between |V'| and |V-V'| (x is in V')
6.     add y to V'
7.     add (x, y) to E'
8. return E'
```

If we use binary heap to implement min priority queue for Prim's algorithm, what is the complexity of finding the next edge to include in the MST (without fixing up the heap)?

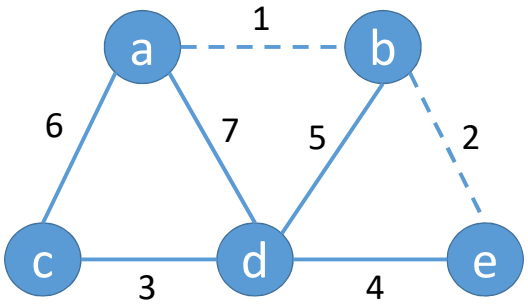
A. O(lg n)
B. O(n)
C. O(1)
D. O(n lg n)
E. $O(n^2)$

Consider the graph below. We are building a Minimum Spanning Tree of this graph. Dashed edges have already been selected by the Kruskal's algorithm to be included in the MST. What is the **next** edge the algorithm will select?



A. (e, d)
B. (a, c)
C. (a, d)
D. (b, d)
E. (c, d)

Consider the graph below. We are building a Minimum Spanning Tree of this graph. Dashed edges have already been selected by the Prim's algorithm to be included in the MST. What is the **next** edge the algorithm will select?



A. (e, d)
B. (a, c)
C. (a, d)
D. (b, d)
E. (c, d)