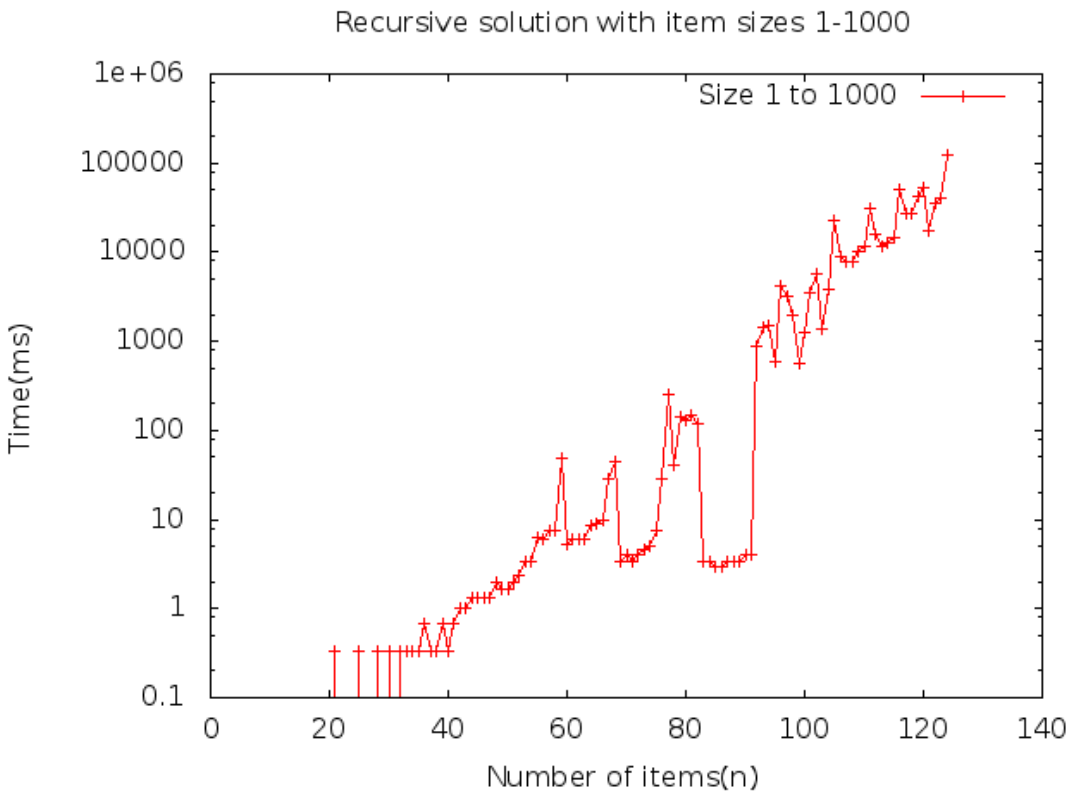
Assignment1 - Knapsack

Alan Christensen - A01072246
January 22, 2014

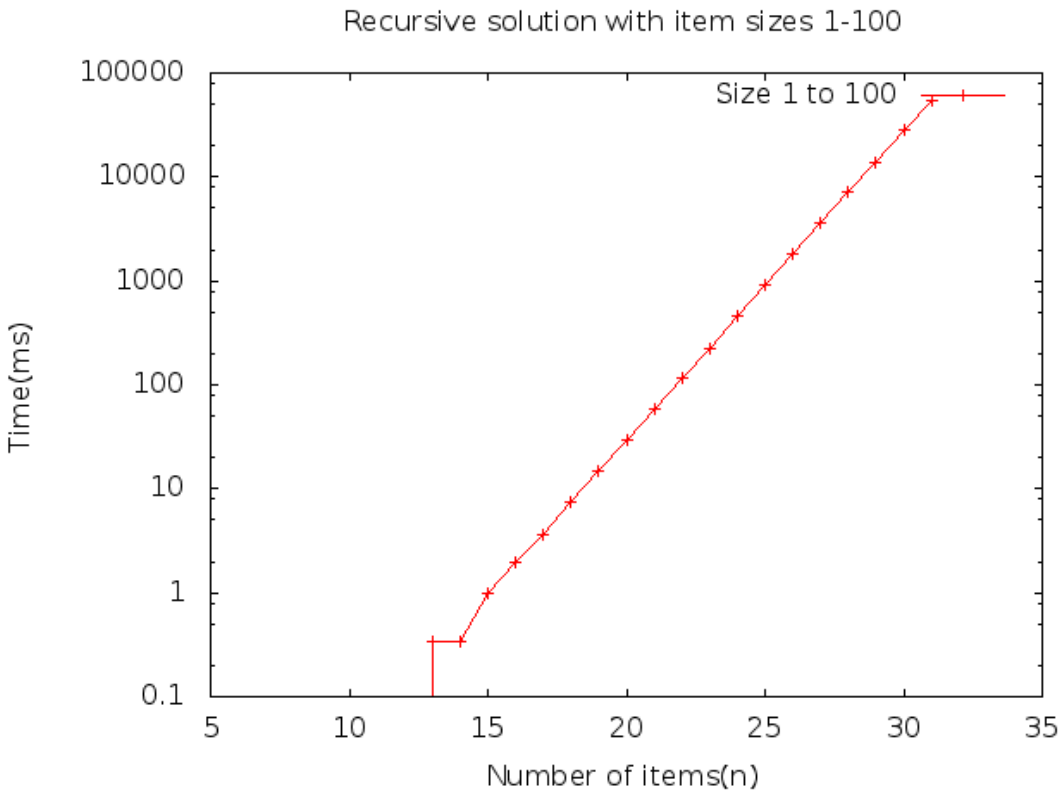
Recursive Solution



This graph shows the recursive algorithm with the Time on the y axis in log scale and the number of items it tried to put in the bag on the x axis in regular scale. SInce the line exponentialy increases the solution is not very efficient. The slope of the line appears to be one log over 20 items. `10^(1/20) ~ 1.1220`

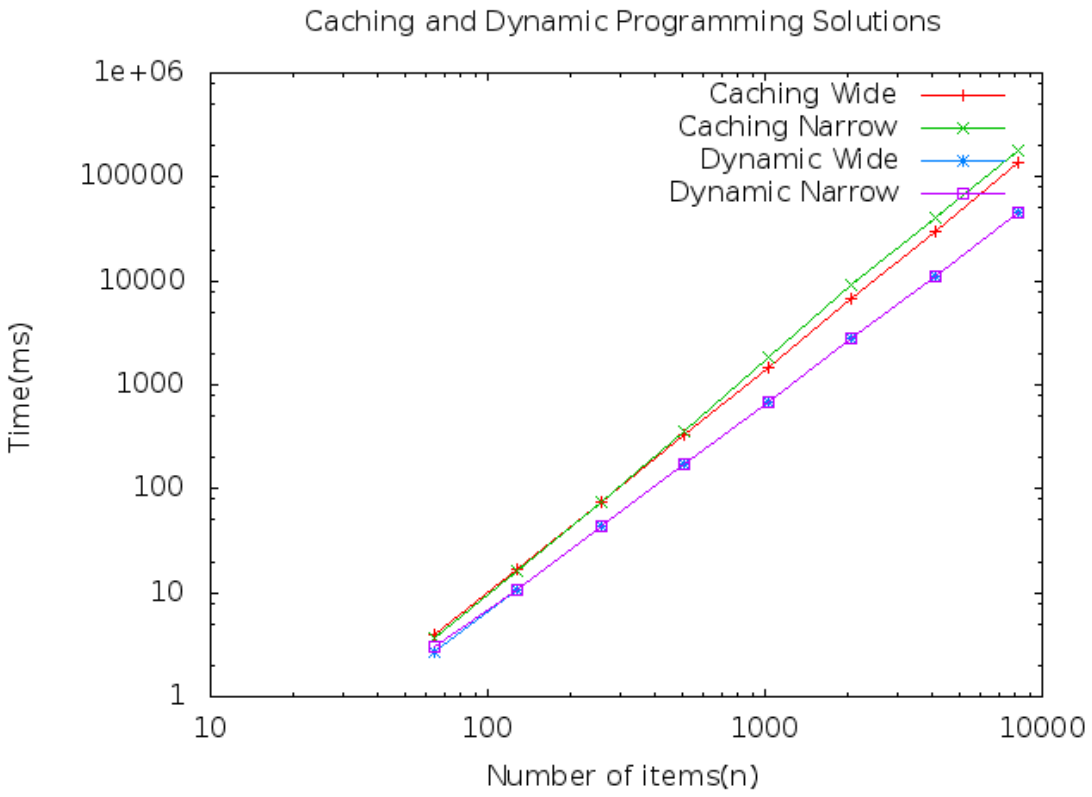
We were expecting that the problem would result in a runtime of about `2^n`. That means that it ran a little faster than our worst case approximation and ran at about `1.12^n`. So don't count on a worst case as actual run time but it is a good approximation and upper bound.

The first graph was using items sizes from 1-1000 and I tested item sizes 1-100 to see the difference and I ended up with a similar result:



This curve ends up more like the worst case. Which it is a worst case. The smaller items always require more calls to fill the bag. With item sizes maxing at 100 it takes 10 max items to fill the 1000 size bag. The curve of this graph is about `10^(1/3) ~ 2`. Meaning we have the 2^n case.

Dynamic Programming and Caching Solutions



I plotted the dynamic and caching together to see which is faster and the results show that it is very similar. This also shows the wide range of sizes verses the narrow range of sizes. All of the solutions are so similar on this graph. It seems in the caching the narrow graph better as the number of items increase. I would say this is because we have less items to cache. As we try the narrow items it is more probable that we would try items of the same size or that combinations of items would hit the same part of the cache. The graph seems to show that this could have happened.

The dynamic solution must fill the entire array every time so the wide vs. narrow range or sizes makes no apparent difference. The runtime depends completely on the size of the cache.

Raw Data

Recursive

Bag Size: 1000
Runs: 30

n	time(ms) 1000	time(ms) 100	n	time(ms) 1000	time(ms) 100	n	time(ms) 1000	time(ms) 100
6	0	0	46	1		86	3	
7	0	0	47	1		87	3	
8	0	0	48	2		88	3	
9	0	0	49	2		89	3	

10	0	0	50	2		90	4	
11	0	0	51	2		91	4	
12	0	0	52	2		92	865	
13	0	0	53	3		93	1,440	
14	0	0	54	3		94	1,503	
15	0	1	55	6		95	602	
16	0	2	56	6		96	4,233	
17	0	4	57	8		97	3,233	
18	0	7	58	8		98	1,930	
19	0	15	59	50		99	558	
20	0	30	60	5		100	1,232	
21	0	59	61	6		101	3,434	
22	0	115	62	6		102	5,718	
23	0	225	63	6		103	1,403	
24	0	459	64	9		104	3,820	
25	0	906	65	9		105	23,224	
26	0	1,808	66	10		106	8,949	
27	0	3,596	67	29		107	7,942	
28	0	7,218	68	44		108	7,909	
29	0	13,858	69	3		109	10,296	
30	0	27,992	70	4		110	11,740	
31	0	53,455	71	3		111	31,287	
32	0		72	4		112	16,008	
33	0		73	5		113	11,606	
34	0		74	5		114	12,709	
35	0		75	8		115	14,664	
36	1		76	29		116	50,501	
37	0		77	255		117	27,346	
38	0		78	41		118	27,611	
39	1		79	143		119	42,892	

40	0		80	130		120	53,576	
41	1		81	150		121	17,157	
42	1		82	118		122	36,204	
43	1		83	3		123	41,299	
44	1		84	3		124	125,932	
45	1		85	3				

The columns with time(ms) 1000 were tested with items size 1-1000
The columns with time(ms) 100 were tested with items size 1-100

Caching and Dynamic

n (#items)	Caching Wide	Caching Narrow	Dynamic Wide	Dynamic Narrow
64	4	3.66667	2.66667	3
128	16.6667	16.3333	10.6667	10.6667
256	74	73.6667	43	43.3333
512	326.667	357.667	174	173.333
1024	1483.67	1813.67	694.333	690
2048	6714.33	9165.33	2779.33	2765.33
4096	30419.7	40860.7	10939	11065.3
8192	136268	177094	44735	44815.3